AFIT/GCE/ENG/92D-01

PARALLEL SIMULATION OF STRUCTURAL VHDL CIRCUITS ON INTEL HYPERCUBES

THESIS

Thomas A. Breeden
Captain, USAF

AFIT/GCE/ENG/92D-01

AFIT/GCE/ENG/92D-01

PARALLEL SIMULATION OF STRUCTURAL VHDL CIRCUITS ON INTEL HYPERCUBES

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Thomas A. Breeden, B.S.E.E

Captain, USAF

Dec, 1992

Approved for public release; distribution unlimited

Acknowledgements

I'd like to thank my thesis committee for their enthusiastic support and guidance. Maj Kanzaki kept my nose to the schedule, and Maj Christensen was always available when I needed to iron out an issue or bounce an idea off of a "greater mind." Dr. Hartrum was especially helpful in every area of this effort. He has an uncanny ability to create a workable model of ***anything*** on his chalkboard. After a session of "chalk-talk," I could always go back to the computer and implement our design in a straightforward manner. Even more uncanny is Dr. Hartrum's method of treating you like an equal and making you feel like what you're doing is truly worthwhile—a highly effective management technique that I hope to take with me.

This research could not have been accomplished without a significant amount of technical support from the following people:

- Ron Comeau demonstrated that extracting intermediate C code from Intermetrics' VHDL compiler was an effective method for generating parallel VHDL simulations. He is responsible for identifying the methods for transforming the C code, the key data structures, and the basic sequential simulation algorithm. From there, I went my own way; however, my "successes" could not have been achieved without his initial investigation and design. Also, the VHDL source code for the adders were taken from Comeau's research.
- Dave Daniel provided the VHDL source code for the shifter.
- Maj Kanzaki created the VHDL wallace tree multiplier, which is the largest structural circuit simulated.
- Scott VanHorn brought me up to speed on SPECTRUM in less than a week. He was also a great help in designing that very tricky receive filter.

- Maj Christensen has begun work on a "VHDL graph tool," which will be used in the future to generate and test circuit partitioning strategies. I used this tool to generate the uniform random distributions of behaviors to logical processes for the shifter and multiplier.
- Rick Norris provided a tool to automatically generate `lp.arcs` files, which SPECTRUM requires. I found generating these files "by hand" to be the greatest source of user error in running the parallel simulations.

To my best friend and brother, Tim, thanks for keeping me laughing with the e-mail. Good luck at Carnegie Mellon.

Finally, to Barbara and the boys, I present this modest thesis as evidence that I was indeed at school all those nights. Thanks for your patience.

Thomas A. Breeden

Table of Contents

List of Figures

List of Tables

Abstract

Many VLSI circuit designs are too large to be simulated with VHDL in a reasonable amount of time. One approach to reducing the simulation time is to distribute the simulation over several processors. This research creates an environment for designing and simulating structural VHDL circuits on the Intel iPSC/2 and iPSC/860 Hypercubes. Logic gates and system behaviors are partitioned among the processors, and signal changes are shared via event messages. Circuit simulations are run over the SPECTRUM parallel simulation testbed, and the null-message paradigm is used to avoid deadlock. Structural circuits ranging from forty to over one thousand logic gates are correctly simulated. Although no attempt is made to find *optimal* partitioning strategies, speedups are obtained for some configurations.

PARALLEL SIMULATION OF STRUCTURAL VHDL CIRCUITS ON INTEL HYPERCUBES

I. Introduction

1.1 Background.

Advances in Very Large Scale Integrated (VLSI) circuit technology increase the transistor count on a chip by about 25% per year, doubling every three years (17:17). In order to efficiently design increasingly complex VLSI circuits, designers use simulation tools to validate their circuits prior to fabrication. In 1979, the Department of Defense (DOD) started the Very High Speed Integrated Circuit (VHSIC) program to employ the use of high density VLSI circuits in military systems. The VHSIC Hardware Description Language (VHDL) program began in 1983 to standardize the tools needed to efficiently design and test these circuits (13, 22).

Many circuit designs are too complex to be simulated with VHDL in a reasonable amount of time. In an effort to improve VHDL's performance, the Defense Advanced Research Projects Agency (DARPA) has sponsored the QUEST project, whose goal is a thousand-fold speed-up in VHDL simulation (28:1-1). One approach to reducing the simulation time is to distribute the simulation of the design over several processors. If VHDL's capabilities could be effectively mapped to a parallel processor, the simulation would be faster and users could design and run more complex circuits. Efforts at AFIT have centered on creating a parallel implementation of VHDL for this purpose.

In 1991 AFIT investigated the data structures of Intermetrics' sequential VHDL simulator and demonstrated a way to intercept intermediate C code from Intermetrics' compiler, transform it,

and run parallel simulations on the Intel iPSC/2 Hypercube (10). This research effort composes the tools necessary to create and run structural VHDL simulations on the Intel iPSC/2 and iPSC/860 Hypercubes.

1.2 Problem Statement.

AFIT has investigated implementing a parallel VHDL simulator to decrease the simulation times of VLSI circuits; however, an automated method for creating parallel VHDL circuit descriptions, a correct parallel simulator, and a common distributed testbed are necessary to generate and simulate large VHDL circuit models.

1.3 Research Objectives.

The main objective of this thesis is to demonstrate and test the capability of mapping large sequential VHDL circuit descriptions to distributed processing systems. The main goals are to

- automate the procedures for generating hierarchical, structural VHDL models.
- create a VHDL simulator that correctly simulates structural VHDL circuit descriptions and is flexible enough to partition simulations among the processors of a distributed system.
- provide a common testbed to facilitate experimentation with parallel simulation protocols and investigation into optimizing circuit partitioning strategies.
- demonstrate the simulator with several VHDL models.
- determine if speedup can be achieved through the use of parallel simulations.

1.4 Assumptions.

Comeau did the preliminary research into transforming Intermetrics' VHDL models into models that can be simulated in a parallel environment. The following assumptions build upon Comeau's research (10:1-3):

- While strict meanings of "parallel" and "distributed" processing systems vary from source to source, AFIT has generally accepted "parallel processing" to indicate processing on a single computer composed of multiple processors, while "distributed processing" refers to processing among several "independent" computers across a network. Nonetheless, as with Comeau's thesis, this research uses the terms "parallel" and "distributed" interchangeably throughout.
- The parallel computers used for development and research are the Intel iPSC/2 and iPSC/860 Hypercubes.
- Source code is written in the standard C programming language (non-ANSI).
- To further research efforts for both DARPA and AFIT and stay consistent with the AFIT environment, the Chandy-Misra conservative synchronization algorithm for event-driven simulations is used. In this thesis, the null-message protocol is implemented via the use of a parallel simulation environment known as SPECTRUM (Simulation Protocol Evaluation on a Current Testbed using Reusable Modules) (32).
- The output from the analyze, model generate, and build phases of the Intermetrics VHDL compiler are correct and accessible.
- The VHDL test cases are within the VHDL subset that is used to demonstrate parallelized VHDL.
- VHDL source code is compiled and model generated in Intermetrics VHDL, Version 2.1, September 1990.

1.5 Scope.

Comeau outlined ten steps to transform Intermetrics' intermediate C code into modules that can run on a parallel VHDL simulator (10:4-6). These operations are automated, and new steps are added to reduce unnecessary function calls and enhance simulator capabilities.

A new parallel simulator, VSIM, is written. The concepts for VHDL simulation are taken from Intermetrics' simulator, and from Comeau's parallel VHDL simulator called PVSIM. The parallelization of VSIM is accomplished with minimal changes to the application by utilizing SPECTRUM.

The parallel simulation protocol is implemented using SPECTRUM "filters." This provides a level of modularity that aids future experimentation with new protocols and instrumentation.

Various circuits are implemented and tested. Also, feedback among LPs is demonstrated.

1.6 Limitations.

1.6.1 VHDL Source Code Limitations for VSIM. The subset of circuits that can be simulated with VSIM includes structural descriptions of logic gates and other simple processes. Circuits are created the same way as for Intermetrics' circuits, with the following limitations:

Signals can be bits or bit-vectors; however, bit-vector inputs must be described one bit at a time, e.g., `Bus(0) <= '1' after 10 ns;`.

Processes should be one-line descriptions (`Out1 <= In1 AND In2 after gate_delay;`); however, multiline processes—delimited by **begin** and **end process** may be used provided they either wait on all signals, or the process only executes once. For example, if a process has input signals **a**, **b**, and **c**, then the following process declarations are acceptable:

```
process
begin
```

```
   wait on a, b, c;
   -- process description here
end process;
```

```
process(a,b,c)
begin
   -- process description here
end process;
```

```
process
begin
   -- process description here
   wait; -- that is, wait indefinitely
end process;
```

It is uncertain how functions and procedures may act in VSIM. For example, functions to describe multi-valued logic—or bus resolution—have not been implemented or tested.

As in the case with functions, VHDL attributes, "buffer" ports, file I/O, etc., have not been implemented or tested.

1.6.2 Postprocessor Limitations. A postprocessor, **pbuild**, is designed to transform Intermetrics generated intermediate C code for parallel simulation with VSIM. Therefore, the postprocessor only works for Intermetrics-generated intermediate C code.

The postprocessor depends heavily on recognizing unique patterns in the intermediate C code. This is accomplished using **lex**, a UNIX-based lexical analyzer. If future enhancements are to be made to the postprocessor, or if the subset of VHDL circuits is to be expanded, each step of the postprocessor should be re-evaluated for possible impact.

1.6.3 VSIM limitations. The user must first run the parallel simulator on one node to identify behavior id's.[1] This is accomplished by enabling a "**MAPPING**" definition in the simulator. Only then can a circuit-to-process mapping be defined.

Circuit partitioning must be done "by hand," i.e., the user creates the appropriate files to define logical process (LP) relationships and behavior-to-LP assignments.

The "receive message" filter used with SPECTRUM is based on a current filter called "chan-clocks." However, the new filter is modified to have access to the local LP's next event time in VSIM; therefore, the *protocol* (in the SPECTRUM filter) is modified in an ***application specific*** manner.

When **OUTPUT** is defined in VSIM, *every* signal change is reported. This becomes a bottleneck in parallel simulations on Intel Hypercubes, as processors contend for common resources, e.g., the host operating system and the disk drives.

1.7 Thesis Overview.

Chapter 2 analyzes the current research efforts in parallel discrete-event digital simulation and how they relate to this thesis. Also, other efforts in parallel VHDL simulation are reviewed. Chapter 3 provides the methodology for implementation of the post-processor, the parallel VHDL simulator, and enhancements to the parallel VHDL environment. Implementation of this methodology is discussed in Chapter 4. Chapter 5 discusses the research findings and results. Finally, conclusions and recommendations for further research are included in Chapter 6.

In addition, the following appendices are included:

- Appendix A: *Definitions.*

[1] A "behavior" is an executable process representing a VHDL logic gate or other simple process.

- Appendix B: *AFIT Parallel VHDL User's Guide.* Documentation on how to prepare and run VHDL descriptions in the parallel processing environment. Also, a test case is demonstrated—using an edge-triggered D flip-flop.
- Appendix C: *Subset of VHDL Source Code for Parallel Simulation.* Describes, with examples, the subset and syntax for VHDL source that can be simulated with the parallel VHDL simulator.
- Appendix D: *Design of the Wallace Tree Multiplier.*
- Appendix E: *Summary of Performance Data.*

1.8 Summary.

VHDL models are executed sequentially in current commercial simulators. As chip designs grow larger and more complex, simulations must run faster. One approach to increasing simulation speed is through parallel processing. This research transforms the hierarchical structural models created by Intermetrics' sequential VHDL simulator into models for parallel execution on the Intel iPSC/2 and iPSC/860 Hypercubes.

II. Background

2.1 Overview.

In this chapter, several simulation techniques are discussed, including traditional simulation techniques on sequential machines, distributed simulation techniques, and digital logic simulation. Also, previous attempts to parallelize VHDL are reviewed.

2.2 Traditional Simulation.

Many real-world systems can be modeled and simulated, using computers, to study their behavior under various conditions. Examples of simulators include battlefield simulators, flight simulators, simulations of factory assembly lines, electronic circuit simulations, etc. In ***continuous simulations***, the state of the model may change continuously over time. ***Discrete-event simulations*** are used to model processes whose states change discretely at specified points in time, as shown in Figure 1 (27). Continuous systems, like digital circuits, may also be modeled with discrete-event simulations.

Sequential simulators usually utilize three data structures (15):

1. The ***state variables*** which describe the state of the system.
2. An ***event list*** which contains the schedule of all future events.
3. A ***global clock*** variable to maintain the simulation time.

There are two main methods of implementing discrete simulations—***time-driven*** simulations and ***event-driven*** simulations. In time-driven simulations, the global simulation clock is used to advance the simulation uniformly through time. With respect to digital circuits, the time-driven approach is not very efficient. If a circuit is in a quiescent state for a long period of time, waiting for the clock to advance becomes time consuming and reduces performance. In event-driven simulation,

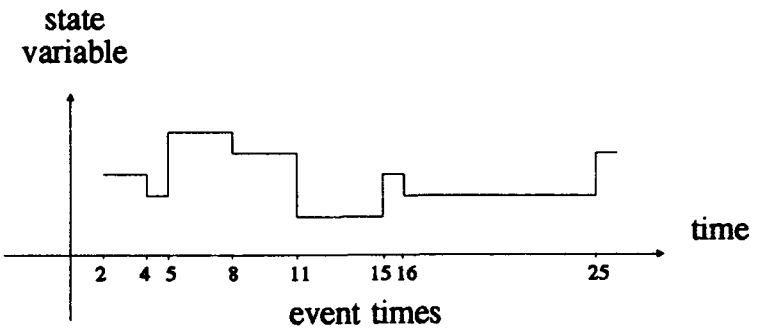


Figure 1. Response Measurement from a Discrete-Event Simulator (27)

processes *schedule* their outputs on a global event list. Then the simulation clock can advance from one event time to the next, since no computations need to occur between event times (26:136-137).

Given this introduction to traditional *discrete-event* simulation, techniques for distributed simulation can be reviewed.

2.3 Distributed Simulation.

With traditional techniques on sequential processors, large simulations in engineering, meteorology, military applications, and circuit design, to name a few, consume large amounts of time (15). Parallel discrete-event simulation, or distributed simulation, refers to the execution of a simulation on a number of processors. Ideal candidates for distributed simulation are systems whose ***physical processes*** (PPs) execute concurrently and can be modeled by message passing among their corresponding ***logical processes*** (LPs) (7:198–199). Electronic circuit systems can be simulated in this way, where the LPs representing the components, or groups of components, that make up the circuit are partitioned among the processors. Hence, the time required to complete a simulation should decrease since computations are executed in parallel (5:11).

2.3.1 General Performance Model. The use of a global clock in distributed simulation constitutes a bottleneck because the LPs would all operate in lock-step. At any global time t, a number of LPs may have nothing to do. In *asynchronous* models, however, each LP contains a local virtual time (LVT), and the LPs are allowed to progress at irregular intervals. In most models, LPs communicate via time-stamped messages in the form of tuples, (t_k, m_k), where m_k is the message sent at LVT t_k (7:199). The specific rules for message passing depend on the particular protocol.

A global event-list would also be a bottleneck in distributed simulation. Therefore, each LP usually maintains its own event-list, or queue. Events either received or self-generated can be scheduled in the local event-list, if necessary, as well as sent to "downstream" LPs, as required by the model (7:198).

2.3.2 Speed-Up and Efficiency of Distributed Simulations. If the simulation time for p processors is T_p, and the time for the same simulation on one processor is T_1, then the *speed-up* of the distributed simulation is T_1/T_p. An ideal speed-up would be p. The *efficiency* of the simulation is therefore the speed-up divided by p. The efficiency indicates how much the communications overhead, time-management, amount of concurrency, and load imbalance among LPs deters the overall speed-up (29:43) (14).

2.3.3 Distributed Simulation Protocols. Asynchronous simulation protocols can be loosely classified as either *conservative* or *optimistic*. Conservative protocols allow an LP to advance its LVT only when it is absolutely certain it cannot receive an event with a time-stamp less than the new LVT. Optimistic protocols allow each LP to proceed at its own pace even though events may arrive out of the past. Time Warp corrects out of order messages by rolling back, i.e., restoring its state to a time prior to the actual message time and then recomputing forward. Therefore, optimistic protocols require state saving capabilities for each LP.

2.3.3.1 Conservative Distributed Simulation Protocols. Chandy and Misra have proposed an asynchronous conservative protocol where each LP manages its LVT and event-list as follows:

> An LP simulates the corresponding PP in the following manner. Let the sequence of messages sent by LPi to LPj be $(t_1, m_1), (t_2, m_2), (t_3, m_3), \ldots$ We require that
>
> 1. $0 \leq t_1 \leq t_2 \leq t_3 \ldots$, (monotonicity) and
> 2. PPi must have sent message m_k to PPj at time t_k, $k = 1, 2, 3, \ldots$ and
> 3. PPi must have sent no other messages to PPj besides $m_1, m_2, \ldots, m_k, \ldots$, i.e., the sequence of messages sent by an LP must correspond exactly to the actual sequence of messages sent by the corresponding PP. During the course of the simulation, if LPi sends LPj a message (t_k, m_k) it implies that all messages from PPi to PPj have been simulated up to time t_k. (7:199)

This model requires static allocation of processes, i.e., the distribution of LPs among the processors is fixed, and the communication paths among the LPs is known prior to simulation (9). Digital circuit simulations, including VHDL, conform to this assumption. This model also assumes no buffering of messages, so a sending LP must wait for all downstream LPs to receive a message before it can progress. Also, an LP must wait for messages from upstream LPs whose clock values are equal to its LVT (7:200).

Misra shows that given this protocol, deadlock can occur in two different ways (25:55). Consider the simple model of Figure 2. Suppose for every message sent by LP0, LP1 generates a message and only sends it to LP2. Then LP4 never receives a message from LP3, because LP3's LVT is still at 0. Therefore, the LVTs of LP4 and LP5 each remain at 0. The other situation that can cause deadlock is cyclic waiting, as shown in Figure 3. The numbers on each arc correspond to the time-stamp of the last message sent. None of the LPs send a message without receiving one first, i.e., they don't predict future messages. LP2 has received a message at $t = 20$ and advanced its LVT to 20, and hasn't generated a corresponding output message (this particular message was *consumed*). So, LP1 is waiting at $t = 20$ to receive a message from LP3, and LP3 is waiting at $t = 15$ for a message from LP2, while LP2 is waiting on LP1. Hence, deadlock has occurred (25, 7).

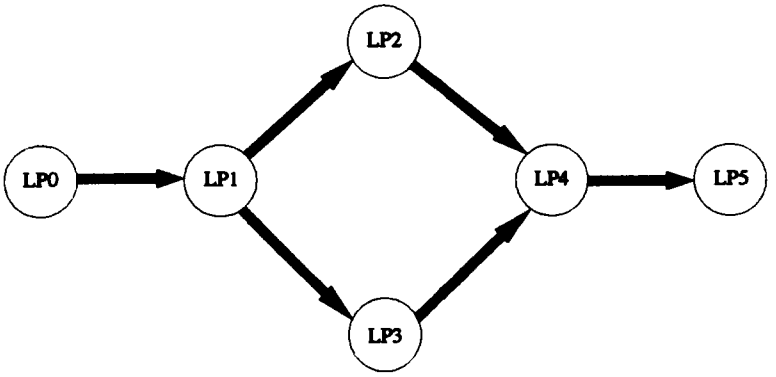


Figure 2. A Distributed System That Does Not Progress (25:56)

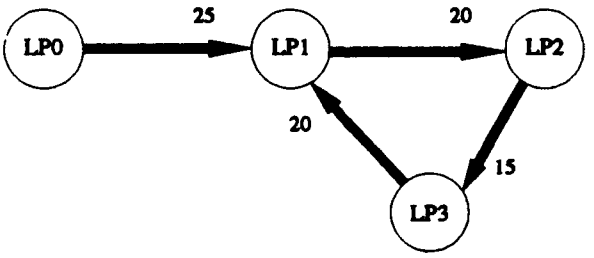


Figure 3. A Distributed System That Deadlocks (25:56)

The two methods for handling deadlocks are *avoidance* and *detection*. Initially, Chandy and Misra proposed the use of ***null messages*** as a means of deadlock avoidance (6). A null message contains just an updated time, and no other state information $(t, NULL)$. The null message guarantees that no messages are sent with a time less than t. In the case of Figure 2, every time LP1 sends a message to LP2, it sends a null message with the same time-stamp to LP3. This allows LP3, LP4, and LP5 to progress. For the cyclic waiting problem of Figure 3, after LP2 receives a message from LP1 at $t = 20$, it sends $(20 + t_{LP2delay}, NULL)$, where $t_{LP2delay}$ corresponds to the propagation delay of LP2. If $t_{LP2delay} = 5$, then LP2 sends $(25, NULL)$, and LP3 responds by sending a null message to LP1 with LP3's propagation delay added to the time-stamp, e.g., $(25 + t_{LP3delay}, NULL)$. In this way, the simulation advances.

The null message approach is costly because a large fraction of messages, and therefore communication overhead, turns out to be null messages (7). Another technique proposed by Chandy and Misra is to allow the simulation—or a subset of the simulation—to deadlock and then use a *master controller* to detect and recover from deadlock (7:202). Detection can be accomplished using the termination detection algorithm of Dijkstra and Scholten, or from a method proposed by Chandy and Misra (8:148) (7:202). Then, the controller polls all LPs that are deadlocked for their earliest next event time. The minimum of these times is the safe time for all deadlocked LPs to advance, since no events can occur before this safe time. Therefore, the controller broadcasts the safe time to all affected LPs, which in turn update their LVTs, and the simulation continues.

The use of a central controller affects simulation performance since it must periodically intervene and evaluate the simulation to see if deadlock has occurred. However, Chandy and Misra maintain the interference is not expected to be a bottleneck since active interference occurs only at deadlock (7:202).

2.3.3.2 Optimistic Distributed Simulation Protocols. The *Time Warp* mechanism for distributed simulation is an *optimistic* protocol. LPs are allowed to go forward in time, risking the

chance that another process may send a message that affects the LP's history. The LP then "rolls back" to the appropriate time in order to handle the new message. This requires each LP to have state-saving capabilities, and to have the ability to "unsend" messages that now are invalid. Also, because of its rollback capability, the Time Warp mechanism can handle dynamic process allocation and connectivity (9). In his survey of parallel discrete-event simulation paradigms, Fujimoto gives the following description of Time Warp:

> The Time Warp mechanism, based on the Virtual Time paradigm, is the most well known optimistic protocol. Here, virtual time is synonymous with simulated time. In Time Warp, a causality error is detected whenever an event message is received that contains a timestamp smaller than that of the process's clock (i.e., the timestamp of the last processed message). The event causing rollback is called a *straggler*. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler, i.e., those processed events that have timestamps larger than that of the straggler.
>
> An event may do two things that have to be rolled back: it may modify the state of the logical process, and/or it may send event messages to other processes. Rolling back the state is accomplished by periodically saving the process's state, and restoring an old state vector on rollback. "Unsending" a previously sent message is accomplished by sending a negative or *anti-message* that annihilates the *positive* messages. If a process receives an anti-message that corresponds to a positive message that it has already processed, then that process must also be rolled back to undo the effect of processing effects of the erroneous computation to eventually be canceled. It can be shown that this mechanism always makes progress under some mild constraints.
>
> As noted earlier, the smallest timestamped, unprocessed event in the simulation is always safe to process. In Time Warp, the smallest timestamp among all unprocessed event messages (both positive and negative) is called global virtual time (GVT). No event with timestamp smaller than GVT will ever be rolled back, so storage used by such events (e.g., saved states) can be discarded. Also, irrevocable operations (such as I/O) cannot be committed until GVT sweeps past the simulated time at which the operation occurred. The process of reclaiming memory and committing irrevocable operations is referred to as *fossil collection*. (15)

2.4 Overview of SPECTRUM.

In 1988, Reynolds recognized the existence of a spectrum of options for parallel simulation protocol designs (30). In order to study *classes* of protocols for *classes* of applications, the University of Virginia developed SPECTRUM (Simulation Protocol Evaluation on a Current Testbed

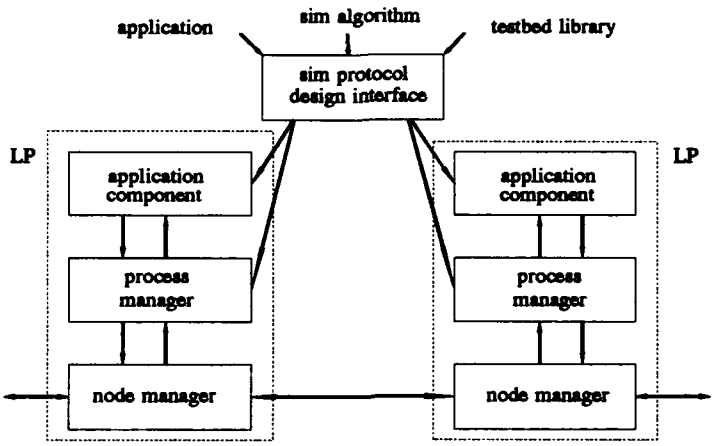


Figure 4. Block Diagram of the SPECTRUM Testbed (31)

using Reusable Modules) (32). SPECTRUM is a common testbed used for creating parallel simulations by taking an application and breaking it into application components, i.e., "pieces" of the application that run concurrently. Each application component, along with a process manager and node manager, make a logical process (LP), as shown in Figure 4. The process manager provides *LP-level* functions to the application for initialization, local clock management, and event handling. The node manager provides hardware-specific functions to the process manager for event traffic among the LPs. To implement specific protocols, filters are written that "intercept" an LP-level function call by the application. The filters may then invoke protocol-specific actions, such as null message generation, LP *polling* for a message, etc.

AFIT has continued to maintain the SPECTRUM testbed as a baseline for queueing simulations (33), battle simulations (3), and VHDL simulations. Also, research is being conducted on a hardware coprocessor that would emulate the basic SPECTRUM functions with microcode capability to modify simulation protocols (11). For details on the SPECTRUM environment at AFIT, refer to Hartrum (16).

2.5 Other Parallel VHDL Research.

In 1989, Proicou (28) developed a distributed system consisting of a scalable kernel that supports VHDL simulations on the Intel iPSC/2 Hypercube. The distributed simulation kernel was an extension of the AFIT VHDL tool set, described in (12). The simulation ran over the SPECTRUM testbed. Proicou found that a general purpose simulation kernel may not be able to take advantage of the presence or absence of feedback loops in the simulation (28:7-1). In general, it was determined unlikely that one distributed kernel design is efficient enough to provide good performance for the wide range of VHDL models. For example, primarily behavioral descriptions may contain a small number of large processes, while primarily structural descriptions may contain a large number of very simple processes (18).

In 1990, Ball and Hoyt (1) reported work in progress to implement a parallel VHDL simulator using "Adaptive Time Warp," in which they look for better performance then Chandy-Misra or Time Warp. Adaptive Time Warp is similar to Time Warp; however, it attempts to reduce "time-faults," i.e., messages that cause roll-back. If a process has recently experienced a high number of "time-faults," it suspends execution for a short time, known as the "blocking window," which is proportional to the message bandwidth. Work is in progress to develop a testbed which implements this strategy.

Comeau's 1991 thesis investigated how to modify a commercial VHDL compiler and simulator—Intermetrics VHDL—for parallel simulation on the Intel iPSC/2 Hypercube (10). In so doing, he looked for parallelism in the intermediate C code generated in the "model generate" phase of compilation. Then, he modified the C code for compatibility with the iPSC/2 and his parallel simulator, PVSIM. PVSIM is the product of a portion of Intermetrics' C source code simulator routines, along with routines added by Comeau. He tested the simulator on three 8-bit adder circuits: a carry-save adder, a carry-lookahead adder, and a ripple-carry adder. In general, simulations using four to

eight processors exhibited a speedup at least twice that of simulations on one node. His results led him to the following conclusions:

- Minimize and balance the number of active signals in a logical process.
- Carefully modify the Intermetrics' generated C source code.
- Ensure a high computation to communication ratio. (10:6-12)

This thesis builds on the lessons learned in Comeau's research. For example, modifying Intermetrics' C code is now automated.

In 1992, Zhang (34) investigated possible methods to partition a VHDL design for hierarchical distributed simulation. He evaluated VHDL entities, blocks, and processes for modularity and concurrency. Zhang reported the following observations with respect to determining the optimal structure (entity, block, or process) for use as an "atomic model" in parallel VHDL simulation (34:203):

- The *entity* descriptions define a clear interface between components; however, using the entity does not fully utilize the inherent parallelism among the blocks and processes.
- The *block* would exploit more parallelism than the entity, but not as much as the process. Also, blocks can be nested, which causes concurrency problems.
- VHDL forces concurrency at the *process* level, so for the greatest amount of parallelism, Zhang concludes that the process is the best atomic model for parallel simulation. However, processes do not have a clearly defined interface—as is the case with entities and blocks.

Zhang introduces the *refined process*, which is generated by defining a *connection port* for every signal or port in a process and removing *wait* statements. In this manner, the process interface is clearly defined. While this method exploits the maximum amount of parallelism and provides a way to *theoretically* study the behavior of a VHDL design, Zhang concludes that it is not robust enough for practical use (34:204).

2.6 Summary.

Discrete-event simulation mechanisms are commonly used in sequential simulations. By introducing the concept of *logical processes*, *local virtual time*, and *message-passing*, asynchronous simulation protocols can extend simulation principles to exploit parallel and distributed computers. The *conservative* Chandy-Misra protocol guarantees an LP does not receive messages out of order with respect to time, but a mechanism must be provided to avoid or detect deadlock. The *optimistic* method of Time Warp allows LPs to proceed at their own pace based on present information. If a message comes in with a time stamp in the past, then an LP must *roll back* to that simulation time in order to handle the message.

As interest in increasing the performance of VHDL grows, a number of research efforts have been conducted to investigate ways to map VHDL simulations to parallel processors. This thesis continues the work initiated by Comeau—mapping Intermetrics' VHDL capabilities to the Intel iPSC/2, and now also to the iPSC/860.

III. Methodology

3.1 Introduction.

When a VHDL circuit is compiled with the Intermetrics VHDL toolset, the intermediate C code can be intercepted and transformed to be linked with AFIT's parallel VHDL simulator (VSIM). VSIM can run sequentially on a single processor, or in parallel on the Intel iPSC/2 or iPSC/860 Hypercubes. For parallel simulations, VSIM runs over SPECTRUM. The subset of VHDL circuits that can be simulated with VSIM includes structural descriptions of logic gates and simple processes. The "behavioral instances" which represent these processes are grouped into logical processes (LPs) and the LPs are distributed among the nodes of any cubesize.

Comeau identified the data structures and the basic cycle required for simulation (10:3-9). This chapter reviews the data structures, simulation cycle, and requirements for parallel simulation.

3.2 Overview.

In order to run a sequential VHDL simulation with the Intermetrics' VHDL toolset, the circuit designer must compile the VHDL source code, and then build and simulate the circuit model. This process is shown in Figure 5.

Circuits are first compiled using the `vhdl` command. This generates an IVAN file (which stands for Intermediate VHDL Attributed Notation). The IVAN file contains the intermediate C code descriptions of the circuit components—which the simulator uses. By using Intermetrics' compiler, the syntax and semantics of VHDL circuit descriptions have already been checked, and correct C code is automatically generated. Normally, generation of the IVAN file is transparent to the VHDL circuit designer.

During the *model generate* phase, the specific C code descriptions—and their header files—are extracted from the IVAN file and object files are created. These files are also normally transparent

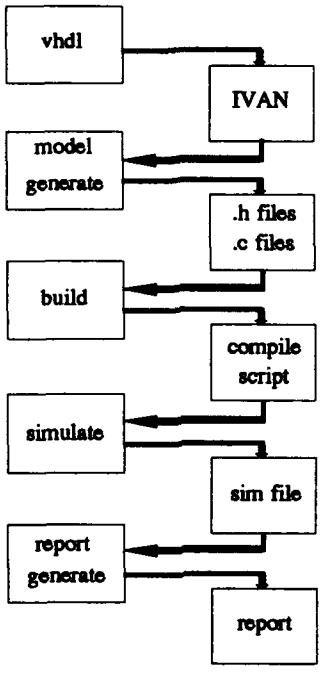


Figure 5. Simulation Session Using Intermetrics' Toolset (10:3-8)

to the designer; however, for parallel simulations, they are transformed into files that are compatible with VSIM.

In the *build* phase, a compilation script is generated that compiles and links the C modules with Intermetrics' simulator modules for operation. Now, the circuit can be simulated with the **sim** command, and a report can be generated with the **rpt** command using Intermetrics' report control language.

For parallel operation, the intermediate C code is transformed into C code that can be linked with VSIM and run on the hypercube, as shown in Figure 6. For this to happen, the code is transformed using a postprocessor called **pbuild**, which reads the compilation script file and uses **plex** to extract and transform the intermediate code. The new code is linked with VSIM, which, together with SPECTRUM, runs the simulation on the hypercube.

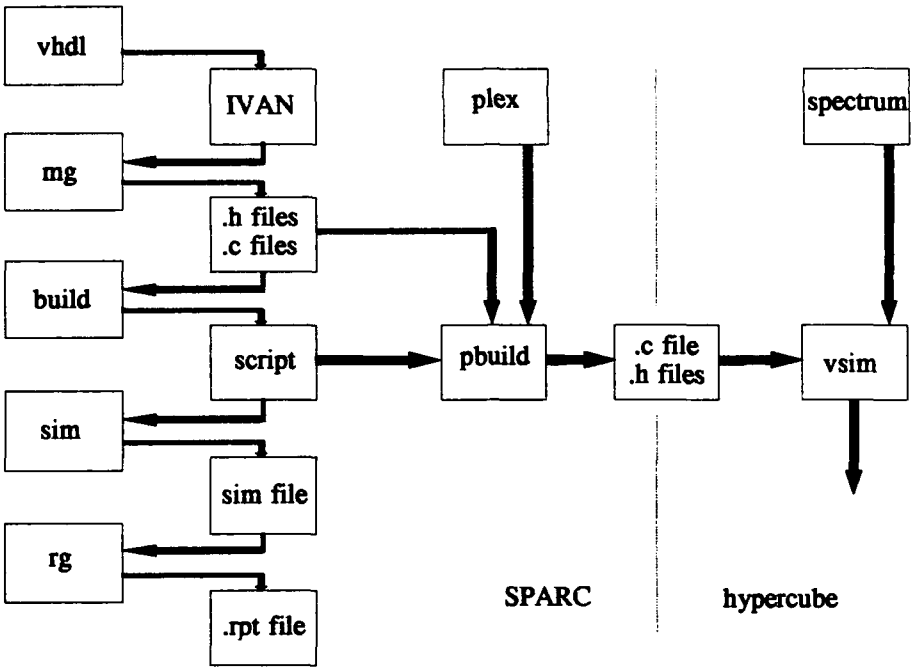


Figure 6. Parallel Simulation Session

3.3 Data Structures.

The data structures for sequential operation are based upon Intermetrics' VHDL simulator. The four main data structures are as follows:

- **Behavior Instances.** The behavior instances are used to describe the behavior of each component (AND gate, OR gate, etc.) and other types of processes. Behavior instances contain a unique id number and a pointer to their execution routine in memory. Several behavior instances may share the same execution routine, e.g., all AND gates in a circuit may use the same algorithm to execute.[1] The basic structure for behavior instances is shown in Figure 7.
- **Signal Records.** The signal records maintain the current state of each signal, including a unique identifier, signal name, current value, size, and pointers to behavioral instances (to

[1] This would be true if, for example, all AND gates used the same entity/architecture pair.

```
typedef struct BHINSTS {              /* behavior instance */
   BHKIND prty;                       /* kind (user, system, etc.) */
   INT32 id;                          /* id */
   ERRT (*exec)();                    /* behavior function */
} BHINST;
```

Figure 7. Basic Structure for Behavior Instances

```
typedef struct {                      /* signal record */
   UINT32 id;                         /* id */
   char *name;                        /* name */
   unsigned size: 4;                  /* size of data value (bytes) */
   UINT32 cval;                       /* current value (offset) */
   CONNT *conns;                      /* behavioral connections */
} SRREC;
```

Figure 8. Basic Structure for Signal Records

identify each signal's connections). The current value field is an offset from a global address space whose base is denoted by the global variable **cv**. See Figure 8 for an example of the signal record structure.

- **Behavior List**. This list contains all behaviors scheduled to execute for the current simulation time. At the beginning of the simulation ($t = 0$), all behaviors are scheduled for execution to initialize their input and output values. As behaviors are executed, they are removed from the list. After the simulation clock advances past zero, signal changes cause affected behaviors to be re-scheduled and re-executed. The behavior list is a simple linked-list called **tmpbeh**, see Figure 9.[2]

[2]The variable name **tmpbeh** is used to maintain consistency with Intermetrics' naming conventions.

```
typedef struct TMPKS {                  /* behavior list */
   BHINST *beh;                         /* behavior instance pointer */
   struct TMPKS *nextb;                 /* next behavior */
} TMPK;
```

Figure 9. Behavior List Structure

```
typedef struct SIG_RECS {               /* active record structure */
   int time;                            /* signal change time */
   SRP sr_ptr;                          /* signal record */
   int value;                           /* possible new value */
   struct SIG_RECS *next_sig_rec;
} SIG_REC;
```

Figure 10. Active Record Structure

- **Active Records.** This is the simulator's next-event list, called `actv`. An "event" corresponds to a behavior output value that *may* be a signal change. Each entry contains an event time, a pointer to the correct signal in the signal record list, and a possible new value for that signal (depending on delay type, etc.), as shown in Figure 10.

An example of the interrelationship of the VHDL data structures is shown in Figure 11. Here, signal number 2, called `CIN`, is changing from a '0' to a '1' at time 50. The active record entry has the new value, and a pointer to the specific signal record. The signal record has a pointer to the global memory space in `cval`, and the list of affected behaviors, i.e., the AND gate and XOR gate. Therefore, these behaviors are added to the behavior list for execution at time 50.

3.4 Sequential Simulation Cycle.

The sequential simulation cycle for VSIM is shown in Fig 12. The following "routines" run

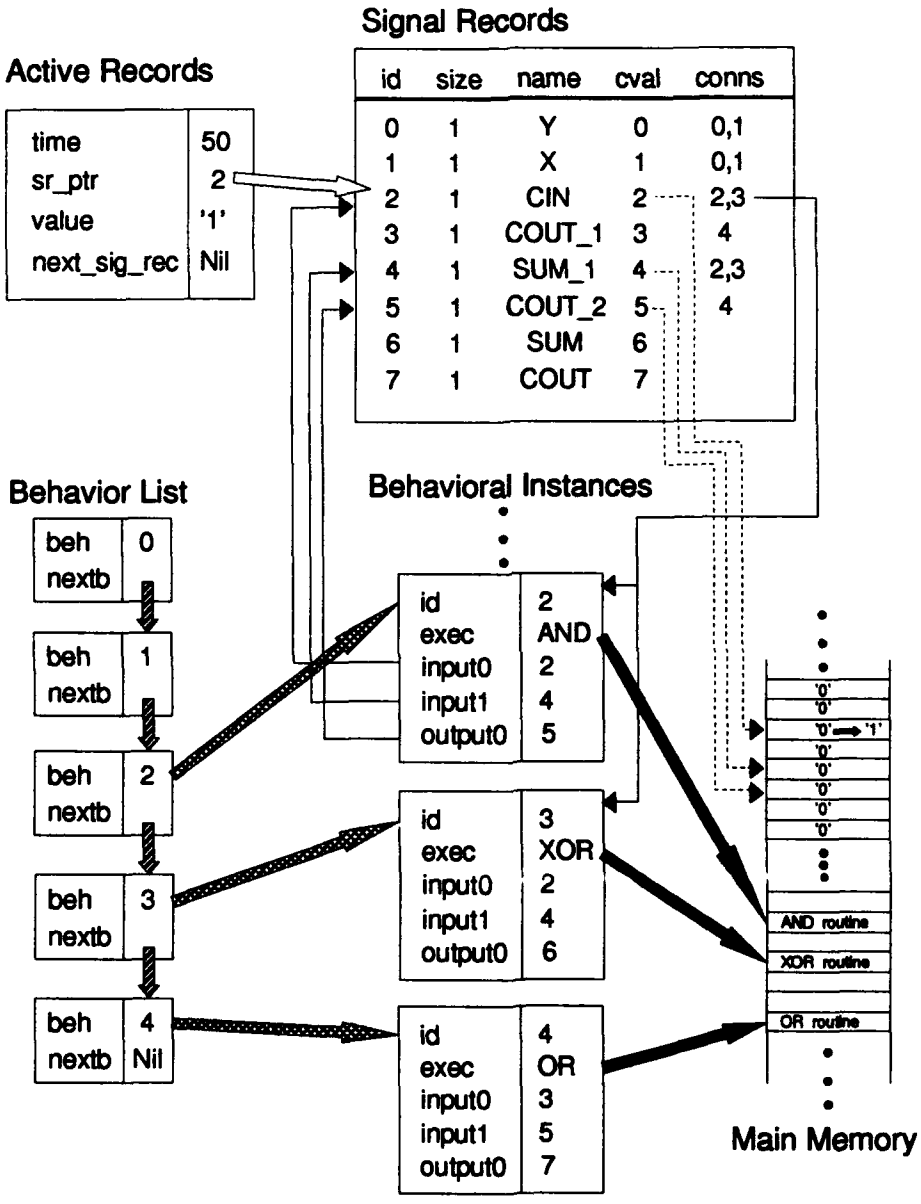


Figure 11. Interrelationship of VHDL Simulation Data Structures (10:3-14)

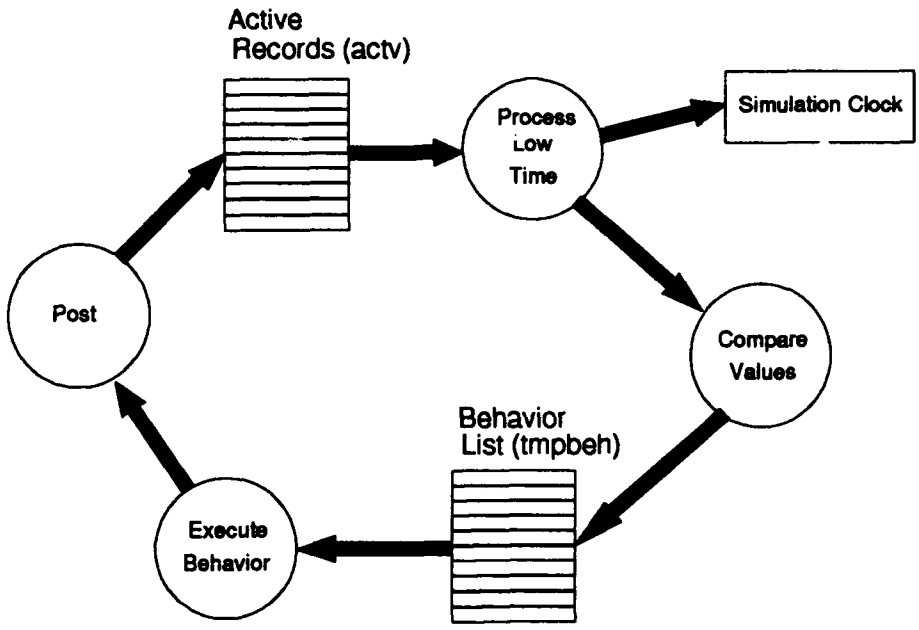


Figure 12. The VHDL Simulation Cycle (10:3-15)

the simulation:

- **post**. Posts each event to the active record list whenever a behavior has executed.
- **get_low_time**. Returns the lowest next-event time from the active records list. The simulation clock is updated to this "low time." Records with this time are removed from the active record list and sent to the **compare_values** routine.
- **compare_values**. Compares the new data value of each event (new to the old data value in memory that is associated with that event's behavior instance, i.e., *circuit component*. If the value is the same, the event is simply ignored (the message is consumed); otherwise, affected behaviors are scheduled on the behavior list for operation.
- **execute_behavior**. Removes behaviors from the behavior list and executes them.[3]

[3] Actual execution of each behavior instance occurs in the intermediate C code. These behavior functions call the **post** function directly.

```
 sim_it ()
{
   SIG_REC *signal;

   /* while active record list and behavior list are not empty */
   while (actv != NULL || tmpbeh != NULL) {
      while (tmpbeh != NULL) {
         execute_behavior();            /* execute behavior and post */
         remove_behavior();
      }
      update_sim_time(get_low_time());  /* process low time */
      while (signal = active_exists(*sim_time)) {
         if (unchanged(signal)) {       /* compare values */
            remove_signal(signal);
         }
         else {
            update_signal(signal);
            schedule_behaviors(signal);
            remove_signal(signal);
         }
      }
   }
}
```

Figure 13. Main Simulation Loop in VSIM

At the beginning of the simulation, input signals are present in the active record list, and all behaviors are scheduled for execution at $t = 0$. The simulation starts at **execute_behavior**. The main (sequential) simulation loop in VSIM is shown in Figure 13. This Figure shows that the simulation cycles from executing behaviors to extracting signal changes until the active list and behavior list are empty. Specifically, while either list is not empty, perform the following:

1. Execute all behaviors on the behavior list, posting the resulting signals after each execution.

2. Update the simulation clock to the next lowest time on the active list.

3. Extract every active record with a time-tag equal to the simulation clock.

4. If the active records indicate a signal change (when compared to their current value in memory), then update the signal's value in memory and schedule affected behaviors.

5. Go back to step 1.

3.5 Active List Management.

A behavior executes at time t when one of its input signals changes at time t. When a behavior executes, the resulting output signal is posted to the active list in time order at $t + t_{DELAY}$, where t_{DELAY} is the delay of the behavior. If n input signals change at t, the behavior executes n times and calls the post routine n times to post the resulting signal output at $t + t_{DELAY}$. Since the behavior executes on each input signal change, the correct output posted to the active list always corresponds to the *last* signal change for a given time, t. Therefore, for correct operation, if an event to be posted matches an event behavior id and time stamp in the active list, the old event is *replaced* by the new event.

In VHDL, a component may be defined to have an *inertial* or *transport* delay-type. An inertial delay corresponds to components which require input signals to persist for a given time before the output signal changes. A transport delay is similar to a "wire delay," the output gets the function of the inputs after delay. The default delay-type for logic gates is inertial.

3.5.1 Transport Delays. Figure 14 shows an AND gate with a transport delay. The output function, *Out_1 = In_1 AND In_2 after gate delay*, occurs regardless of the time duration of the input signals or any combination of input signals. Therefore, no special action is required when posting the output to the active record list.

3.5.2 Inertial Delays. The rule for inertial delays is that the output does not change within the inherent delay of the logic gate. For active list management, if a behavior executes at time t and its corresponding output is to be posted at $t_{NEW\_EVENT} = t + t_{DELAY}$, and a signal change

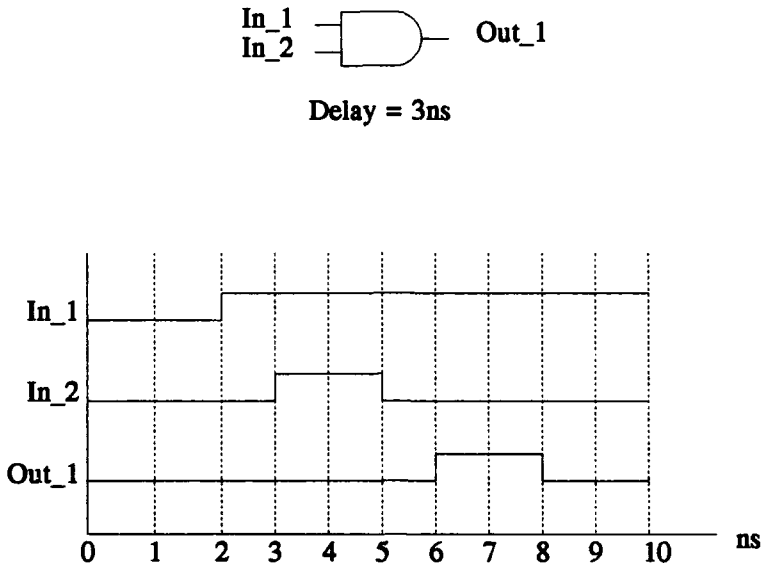


Figure 14. An AND Gate with a Transport Delay.

can be found for the same behavior with a time, t_{EVENT}, where $t < t_{EVENT} < t_{NEW\_EVENT}$, then the output at t_{EVENT} is removed *if the signal value at* t_{NEW_EVENT} *is the opposite of the value at* t_{EVENT}.

Figure 15 shows an AND gate with an inertial delay of $3ns$. At $3ns$, In_2 goes to a logic '1' and the gate is executed. As a result, an output of '1' is scheduled in the active list with a time $t_{EVENT} = 6ns$. At $5ns$, In_2 goes back to '0', the gate is executed, and an output of '0' is generated at $t_{NEW\_EVENT} = 8ns$. When the new event is posted, the change at t_{EVENT} is identified and removed from the active list because $(t = 5) < (t_{EVENT} = 6) < (t_{NEW\_EVENT} = 8)$ and the value at t_{NEW_EVENT} ('0') is the opposite of the value at t_{EVENT} ('1').

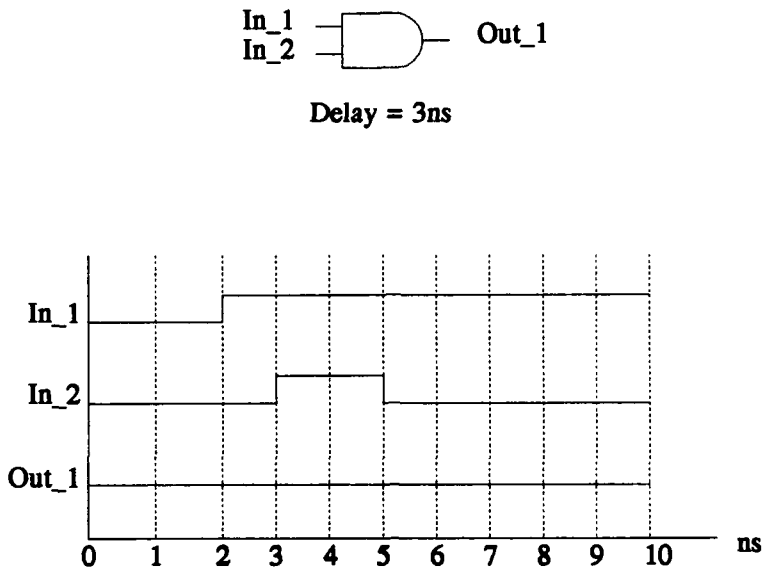


Figure 15. An AND Gate with an Inertial Delay.

3.6 Transformation of Intermediate C Code.

The intermediate C code contains the circuit-specific information. During the simulation, it is this code which instantiates the signals and behaviors, and their interrelationships. Also, this code contains the functions that describe the behavior of every behavior instance.[4]

VSIM does not support every capability of VHDL. For example, processes with wait statements are not supported. Also, complex behavioral processes are not supported, e.g., processes that manipulate integers (instead of bits) as signals. As this project grows, more of the intermediate C code can be included and compiled with VSIM. To make the intermediate C code compatible with the current version of VSIM, the following general steps must be taken:[5]

- Identify and extract the files that were generated during the *model generate* phase.
- Modify the `#include` directives accordingly.

[4] Several behavior instances may share the same function.
[5] The specific steps—and their implementation in `pbuild` and `plex`—are discussed in Chapter 4.

- Remove calls to trace routines and other trace statements. VSIM does not support tracing capabilities.
- Modify the **mksig()** function call to include a field for the signal name. This is so VSIM output can refer to signals by name instead of identifier.
- Modify the behavior functions to report the name of the entity/architecture pair it represents (if **MAPPING** is turned on in VSIM).
- Change **main()** to **vhdl_main()** so VSIM can call it after initialization.
- Modify the intermediate code to call VSIM's **init_cv()** and **sim_it()** routines for circuit initialization and to start the simulation, respectively.

3.7 Parallel VHDL Simulation.

3.7.1 SPECTRUM and VSIM. As shown in Figure 16, VSIM is run over SPECTRUM in order to "parallelize" the simulation and evaluate the effectiveness of various protocols on parallel VHDL simulations while requiring minimal modifications to the original application—VSIM. Spectrum allows the application to be broken into LPs, and the LPs communicate with each other with function calls to the "LP manager"—**lp_man.c**. These function calls can be interrupted by "filters," which may provide additional handshaking, clock, or queue management, as required for various protocols. The main functions are

- **lp_init()**. Ensures LPs are fully initialized. Builds filter tables, if any.
- **lp_get_event()**. Get the next event from the SPECTRUM queue.
- **lp_post_event()**. Send event to specified LP.
- **lp_advance_time()**. Advance an LP's local time.[6]

[6]Recently, a terminate filter was added to SPECTRUM. VSIM was not modified to take advantage of this new filter.

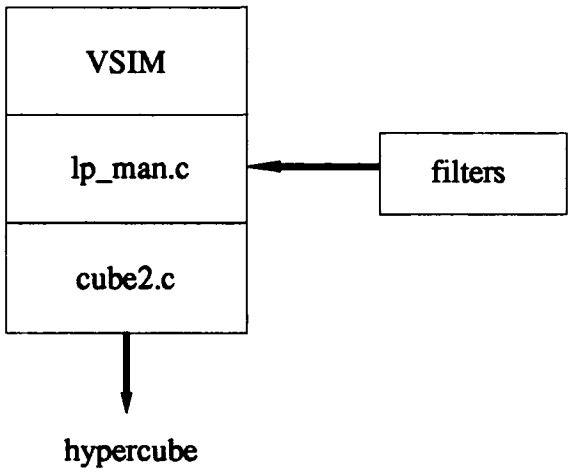


Figure 16. VSIM on the SPECTRUM Testbed (One LP Shown)

The hardware interface to the Hypercubes is provided in the functions in `cube2.c`. In general, `lp_man.c` makes these calls, and the application (VSIM) makes only LP-level calls. LPs can be partitioned among processors in a number of ways. Because of the multitasking capabilities of the Intel 80386, a "logical process" does not have to correspond to a "physical processor." Therefore, a simulation with eight LPs can be partitioned among one to eight processors of the iPSC/2.[7] On the iPSC/860 Hypercube, however, there must be a *one-to-one* mapping of LPs to processors, because each i860 processor does not support multitasking.[8]

3.7.2 The SPECTRUM/VSIM Filters. The SPECTRUM filters for VSIM are based on a previously existing filter called `chanclocks`. These filters provide the null-message protocol.

In general, messages among LPs are signal changes with the structure of Figure 17. Once an event is received, VSIM converts it into an active record and posts it in the active list.

[7] AFIT's iPSC/2 Hypercube has eight Intel 80386 processors.
[8] The iPSC/860 Hypercube at Wright-Patterson AFB has eight Intel i860 processors.

```
typedef struct event {
   int from_lp;  /* lp id of lp sending event */
   int to_lp;    /* lp id of destination lp */
   int time;     /* timestamp of event */
   int event;    /* event type or number */
   int id;       /* signal id */
   int value;    /* signal value */
   struct event *next;
};
```

Figure 17. Event Structure for Message Passing

For this discussion, t_{NULL} is the null message time, t_{QUE} is the lowest time stamp of an LP's SPECTRUM input queue, t_{NEQ} is the "low time" in the local LP's active list (in VSIM), and t_{DELAY} is the output delay of an LP.[9]

The safe time, t_{SAFE}, is the local virtual time (LVT) an LP can safely approach. It is the minimum input time of all input arcs. In other words, an LP knows it does not receive a message prior to this time, so it is safe to advance it's LVT to t_{SAFE}. Incoming NULL messages are used to update this safe time, and serve no other purpose.

Incoming events in SPECTRUM's queue are stored in time order. Therefore, if an event at the head of this queue has a time stamp less than or equal to t_{SAFE}, the event may be passed to VSIM upon request. This is called a "valid event," because by the Chandy-Misra paradigm, it is guaranteed that no messages are received prior to t_{SAFE}.

3.7.2.1 Rules for Null Messages. Null messages are used to avoid deadlock, as discussed in Chapter 2. They are sent from an LP in three cases:

1. Upon initialization, every LP sends a null message at time $t_{NULL} = (0 + t_{DELAY})$.

[9]Strictly speaking, there is a unique output delay for every output arc of an LP, but for this thesis, it is assumed all output delays on each arc are the same.

2. When a signal is sent to another LP via an output arc at time t, all other output arcs are sent a null message at time t.

3. When VSIM requests a signal and SPECTRUM has a valid event, it is returned to VSIM. If there is no event ready, the receive filter checks to see if $t_{NEQ} \leq t_{SAFE}$. If so, a NULL pointer is returned and VSIM continues. Otherwise, the filter waits—or blocks—for an incoming event. When an LP is about to block, it sends a null message at $t_{NULL} = min((t_{SAFE} + t_{DELAY}), t_{NEQ})$ to all downstream LPs. Therefore, deadlock is avoided because every LP sends a "guarantee" that no messages are sent prior to t_{NULL}, and every downstream LP can update their safe times; therefore, cyclic waiting does not occur.

3.7.3 Modifications to VSIM for Parallel Simulation. The VHDL simulation can be partitioned in a number of ways. One method would be to allow each LP to share the behavior instances, but partition the signals among the LPs. When a behavior executes, the LP determines the owner of the resulting signal, and an event is sent to the corresponding LP. Another method—and the one implemented in this research—is to allow the LPs to share signals, but partition the behaviors. This way, only *valid* signal changes are sent to other LPs. When a signal does change, this event is sent to all LPs with affected behaviors. The behavior list of any LP would consist of only behaviors "owned" by that LP. Messages are introduced into the simulation cycle as shown in Figure 18. This cycle is the based on the sequential simulation cycle of Figure 12; however, signal changes that affect other LPs are now sent to those LPs as events. Similarly, after local behaviors are executed and posted, if any upstream events are forthcoming, they are posted in the active record list. Each LP runs the same simulation, but with different data in terms of behaviors. This is known as a single program/multiple data (SPMD) configuration (21).

A parallel simulation in a 2-LP configuration is shown in Figure 19. This Figure shows the connectivity if each LP had signal changes that affected behaviors on the other LP. Another possible configuration for 2-LPs could be that only one LP depended on the other, "upstream" LP.

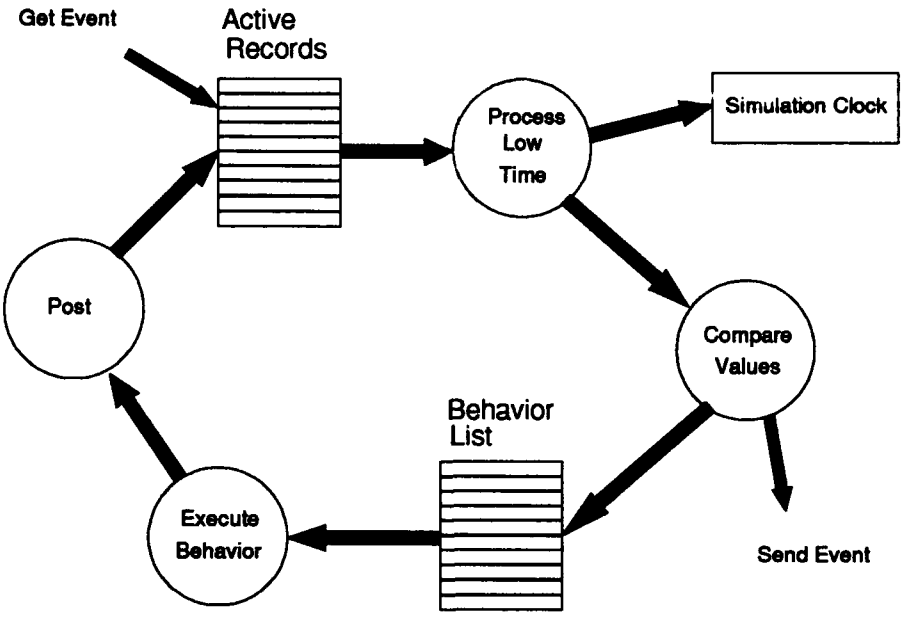


Figure 18. Parallel VHDL Simulation Cycle Shown for One LP

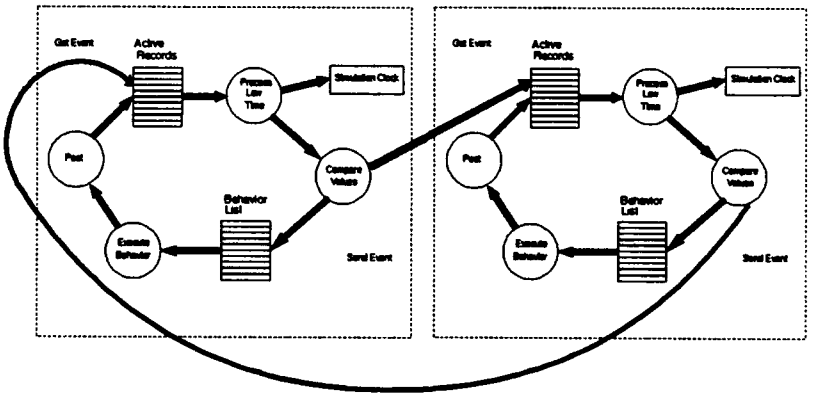


Figure 19. A 2-LP configuration

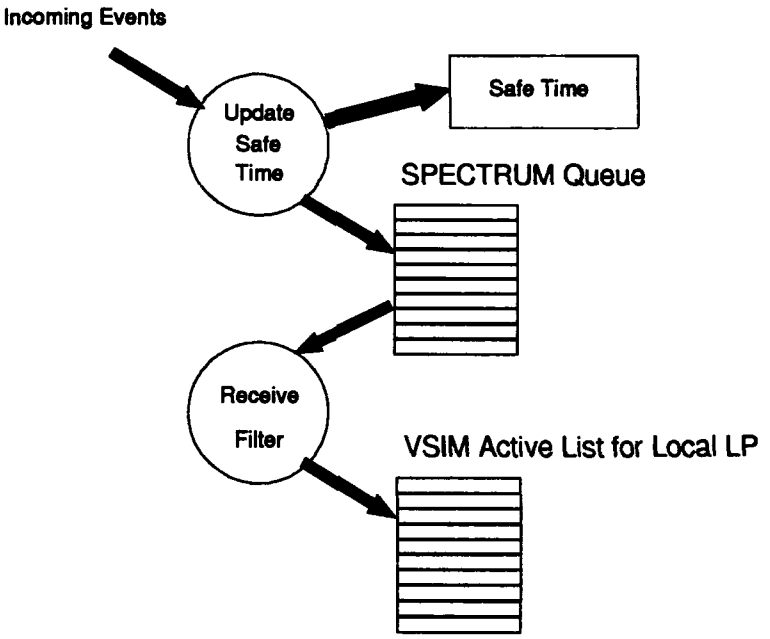


Figure 20. Data Flow for Incoming Event

The process of receiving an event is shown in Figure 20. Incoming events are stored by SPECTRUM in an input queue until requested by VSIM. When SPECTRUM receives events, the input safe time is updated. When VSIM requests an event, the receive filter removes it from the SPECTRUM queue (according the the rules for null messages in the previous section) and passes it to VSIM. In turn, VSIM posts it in it's local active list and continues the simulation.

The main simulation loop of VSIM must be modified to accommodate parallel operation. In sequential operation, the simulation is complete when the active list and behavior list are both empty. This may not be the case for parallel operation. One LP may have empty active and behavior lists, but an upstream LP may send another active record (signal change) to be put in the empty active list. Therefore, each LP must run until the maximum simulation time is reached, as shown in Figure 21.. In support of this change, the `get_low_time()` function is modified to return the maximum time if the active list is empty. This method is correct for parallel and sequential operation.

```
 sim_it ()
{
   SIG_REC *signal;

   while (*sim_time < MAXTIME) {
      while (tmpbeh != NULL) {
         execute_behavior();            /* execute, and post */
         remove_behavior();
      }
      get_signal();                     /* get from other LP and post */
      update_sim_time(get_low_time());  /* process low time */
      while (signal = active_exists(*sim_time)) {
         if (unchanged(signal)) {       /* compare values */
            remove_signal(signal);
         }
         else {
            update_signal(signal);
            schedule_behaviors(signal); /* including sending to other LPs */
            remove_signal(signal);
         }
      }
   }
   end_sim();
}
```

Figure 21. Main VSIM Simulation Loop Modified for Parallel Operation

```
int lp_own[MAX_BEHAVIORS];          /* node location of each behavior */
```

Figure 22. Structure Identifying LP Ownership of Each Behavior

Figure 21 is also modified to do a **get_signal()** after all behaviors have executed for a given time. This function calls **lp_get_event()** from SPECTRUM, converts the event into an active record, and posts the new record into the active list. The **send_signal()** routine is called from the **schedule_behaviors()** function. This way, as behaviors are scheduled on the local LP, it can check to see which other LPs have behaviors dependent on the signal change. The **send_signal()** function, in turn, builds an event out of the signal change and calls SPECTRUM's **lp_post_event()**.

Because each LP must know which behaviors it owns, a few modifications to VSIM data structures must be made. VSIM is modified to read in a mapping of behaviors to LPs, and each LP has this information in the array shown in Figure 22. In order to generate this mapping file, the user must determine the behavior numbers and dependencies. To do this, VSIM is run in sequential mode with **MAPPING** defined in its header file. The corresponding output is run through a program called **vmap**, which generates a list of behavior numbers, names, delays, and dependencies. The user can then use this data to specify which behaviors are grouped to which LPs.[10] The specific LP to processor configuration is defined at run time.

Also, the signal record structure is modified to contain an "ownership" flag, as shown in Figure 23. Since there is a one-to-one correspondence between behaviors and their signal ***outputs***, after behaviors are executed and the corresponding signals records are created, the ownership flag

[10]This procedure is currently done manually, unless a random assignment of behaviors to LPs is used.

```
typedef struct {                    /* signal record */
   UINT32 id;                       /* id */
   char *name;                      /* name */
   unsigned size: 4;                /* size of data value (bytes) */
   UINT32 cval;                     /* current value (offset) */
   CONNT *conns;                    /* behavioral connections */
   BOOL i_own;                      /* for LP ownership */
} SRREC;
```

Figure 23. Basic structure for Signal Records Modified to Identify LP Ownership

is set to **TRUE** for that LP. LPs are responsible to send and/or report signal changes for those signals that they "own."

3.8 Summary.

VHDL circuits are compiled with the Intermetrics VHDL toolset, and intermediate C code is intercepted and transformed to run with AFIT's parallel VHDL simulator. VSIM runs either sequentially on a single processor, or in parallel on the Intel iPSC/2 or iPSC/860 Hypercubes. For parallel simulations, VSIM runs over the SPECTRUM testbed. This allows various protocols to be tested by changing filters instead of making significant modifications to VSIM. Behavioral instances are grouped into LPs and the LPs are distributed among the Hypercube's processors.

This chapter identified the key data structures, the simulation cycle, and the methodology for breaking VHDL simulations into multiple LPs and running on multiple processors.

IV. Implementation

4.1 Introduction.

This chapter describes the implementation of the postprocessor functions **pbuild** and **plex**, and the VSIM interface to the Intel Hypercubes with SPECTRUM. Also, implementation of the null-message protocol using SPECTRUM filters is discussed. For examples of some of the key source code that realizes this implementation, refer to Appendix G.

4.2 Postprocessor Implementation.

As shown in Table 1, even small VHDL circuit simulations are composed of thousands or tens of thousands of lines of C code just for circuit description, i.e., not including simulator code. Large, flat structural descriptions lead to very large intermediate files. It is better to build structural circuits *hierarchically* and use a number of intermediate configuration descriptions than to use one overall configuration file. The multiplier in Table 1 is configured hierarchically, while the shifters are configured as one large structural description. Even though the multiplier has three times as many gates as the 16-bit shifter, the intermediate code is 37% smaller.

In order to decrease the amount of time required to transform this code into code compatible with VSIM, a program called **pbuild** is created to automate this process.

Table 1. Length of Intermediate C Code Circuit Descriptions

Simulation	File Size (bytes)	Lines of Code
SR flip-flop	32679	1304
edge-triggered D flip-flop	41063	1964
full adder	61155	2350
8-bit carry save adder	639757	26651
8-bit carry lookahead adder	569576	23106
8-bit ripple carry adder	504540	20700
8 X 8 wallace tree multiplier	564956	22032
16-bit bit/byte shifter	900307	34192
32-bit bit/byte shifter	1603124	59967

Pbuild reads the compilation script file generated during the *build* phase, concatenates the C files that make up the specific simulation, and calls **plex**, which transforms the data by the rules specified by Comeau (10:4-6) and in this thesis.

The user must determine the name of the build script generated during the build phase. This can be accomplished by adding the Intermetrics debug switch **-debug=cknd** to the **mg** and **build** commands. Then, after the build phase completes, the script filename is reported. For the example of Figure 24—a model-generate and build session for an edge-triggered D flip-flop discussed in Appendix B.3—the compilation script is **FN23309**.

An example build script (for the edge-triggered D flip-flop) is shown in Figure 25. This script is used to generate an executable simulation called **FN23307**, and located in **/home/inter/shiplib/tbreeden**. This directory represents where files in the user's **work** library are located. The intermediate C files required for VSIM are the main (**FN23311.c**), and the **.c** files that correspond to the **.o** files in the **work** directory. For each **.o** file in **/home/inter/shiplib/tbreeden** of Figure 25, the corresponding **.c** filename is "two greater" than it's **.o** file. For example, the **.c** file that corresponds to **FN23304.o** is **FN23306.c**. The program **pbuild** reads this script, recognizes the work library's main and object files, and concatenates the corresponding main and **.c** files, as shown in Figure 26. From this point, **pbuild** calls **plex** for data transformation. If the specific path to the build script is not specified by the user, **pbuild** can determine it by getting the UNIX environment variables **VHDL_LIBROOT** and **LOGNAME**, which in this case would return **/home/inter/shiplib** and **tbreeden**, respectively.[1] This works *as long as models are compiled and model generated in the user's work directory*, otherwise the user may have to specify the complete path to the build script on the command line when invoking **pbuild**.

After extracting and concatenating the correct files, **pbuild** calls **plex** via the operating system, also shown in Figure 26. The **plex** program was created using C and a UNIX program

[1]**The path /home/inter/shiplib is the explicit path on lovelace in the VLSI lab. A logically equivalent path is /usr/vhdl/shiplib, which works on any machine in the VLSI lab.**

```
lovelace.~/vhdl/etdff>mg '-debug=cknd nand_gate(simple)'
Object_file : /home/inter/shiplib/tbreeden/FN23067.o
H file      : /home/inter/shiplib/tbreeden/FN23068
C file      : /home/inter/shiplib/tbreeden/FN23069.c
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

lovelace.~/vhdl/etdff>mg '-debug=cknd three_input_nand_gate(simple)'
Object_file : /home/inter/shiplib/tbreeden/FN23077.o
H file      : /home/inter/shiplib/tbreeden/FN23078
C file      : /home/inter/shiplib/tbreeden/FN23079.c
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

lovelace.~/vhdl/etdff>mg '-debug=cknd etdff(structural)'
Object_file : /home/inter/shiplib/tbreeden/FN23289.o
H file      : /home/inter/shiplib/tbreeden/FN23290
C file      : /home/inter/shiplib/tbreeden/FN23291.c
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

lovelace.~/vhdl/etdff>mg '-debug=cknd etdff_test_bench(structural)'
Object_file : /home/inter/shiplib/tbreeden/FN23299.o
H file      : /home/inter/shiplib/tbreeden/FN23300
C file      : /home/inter/shiplib/tbreeden/FN23301.c
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

lovelace.~/vhdl/etdff>mg '-debug=cknd -top etdff_config'
Object_file : /home/inter/shiplib/tbreeden/FN23304.o
H file      : /home/inter/shiplib/tbreeden/FN23305
C file      : /home/inter/shiplib/tbreeden/FN23306.c
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

lovelace.~/vhdl/etdff>build '-debug=cknd -replace -ker=etdff etdff_config'
Kernel com file is /home/inter/shiplib/tbreeden/FN23309
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
```

Figure 24. Example VHDL Model-generate And Build Session for an Edge-triggered D Flip-Flop

```
#!/bin/csh
if ( $?VHDL_LIBSIM == 0 ) then
  if ( ! -e /usr/local/lib/libsim.a ) then
    echo NOLIB > bld_5854cnl.log
    exit 1
  endif
  setenv VHDL_LIBSIM -lsim
else if ( ! -e $VHDL_LIBSIM ) then
  echo LIBSM > bld_5854cnl.log
  exit 2
endif
cc -g -o /home/inter/shiplib/tbreeden/FN23307 \
/home/inter/shiplib/tbreeden/FN23311.c \
/home/inter/shiplib/tbreeden/FN23304.o \
/home/inter/shiplib/tbreeden/FN23077.o \
/home/inter/shiplib/tbreeden/FN23067.o \
/home/inter/shiplib/tbreeden/FN23289.o \
/home/inter/shiplib/tbreeden/FN23299.o \
/usr/vhdl/shiplib/std/FN240.o \
/usr/vhdl/shiplib/std/FN235.o \
/usr/vhdl/shiplib/std/FN225.o \
/usr/vhdl/shiplib/std/FN25.o
$VHDL_LIBSIM -lcurses -ltermlib -lm -lc >& bld_5854cnl.log
exit $status
```

Figure 25. Example Compilation Script Generated During Intermetrics' Build Phase

```
lovelace.~/vhdl/etdff> pbuild FN23309 etdff.c
cp /home/inter/shiplib/tbreeden/FN23306.c big_etdff.c
cat /home/inter/shiplib/tbreeden/FN23079.c >> big_etdff.c
cat /home/inter/shiplib/tbreeden/FN23069.c >> big_etdff.c
cat /home/inter/shiplib/tbreeden/FN23291.c >> big_etdff.c
cat /home/inter/shiplib/tbreeden/FN23301.c >> big_etdff.c
cat /home/inter/shiplib/tbreeden/FN23311.c >> big_etdff.c
plex < big_etdff.c > etdff.c
Transformation in progress...
```

Figure 26. Result of Reading Compilation Script by `pbuild`

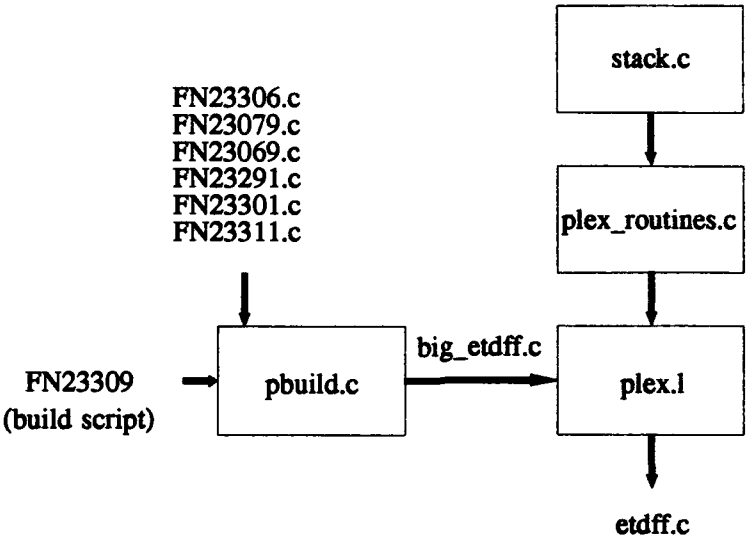


Figure 27. Relationships of the Postprocessor Files

called **lex**. Lex reads a specification file containing UNIX regular expressions and C routines that are associated with the regular expressions. When lex reads the file, character patterns are matched by the rules of the specified regular expressions, then C routines are called that manipulate the input file. For more information on lex, see (23).

Figure 27 shows the necessary files and relationships among them for the complete postprocessor. The files relate to the edge-triggered D flip-flop examples of Figures 24, 25, and 26. The user only has to invoke **pbuild**, which controls the transformation process. The files **plex_routines.c** and **stack.c** are used by **plex** to manipulate the data once a regular expression has been recognized.

4.2.1 Transformation Steps. Pbuild transforms the Intermetrics' compiler-generated **.c** files into a single **.c** file that, along with the associated header files, can be transferred to the iPSC/2

or iPSC/860 and run with VSIM. The following steps are taken to transform the intermediate C code:[2]

1. The intermediate C code representing the VHDL configuration file is brought in first, and it contains all the necessary **#include** directives.[3] Therefore, all **#include** directives after this file are removed.

2. All lines containing **#include fn26** or **#include FN26** are deleted. The necessary header information for VSIM simulations is combined in **vsim.h**, which is already on the hypercube.[4]

3. All remaining **#include** directives are changed to the proper path. For example, if the path was **/home/inter/shiplib/tbreeden/FN2858**, it is changed to **FN2858**.

4. All lines that contain "**{trace**" are changed to "**{**", i.e., "**trace**" to the end of line is deleted. VSIM does not support tracing capabilities.

5. Each occurrence of **if(trceqp) { ... }** is deleted. These **if** statements contain code used with Intermetrics simulator when it's in the "trace" mode.

6. To complete the removal of trace-related statements, every line containing the strings "**trace**" or "**TRAREC**" is deleted.

7. The last function call from the main routine is **Z5xxxxxx** (where **xxxxxx** can be any series of numbers and letters). The statement "**cv = init_cv();**" is inserted before the first line in this function. This new function call (**init_cv()**) is used to perform initialization functions for the parallel simulator. In the third line of the same function the statement "**sim_it();**" is added. This routine starts the simulation.

---

[2]For examples of how steps 1 through 10 are implemented, see Figures 4.4 through 4.17 of Comeau's thesis (10).

[3]This is not true for structural models created hierarchically with a number of configuration descriptions. For such models, the user must add the include directives to the **ckt.c** file. The files to include can be found by examining the **big-ckt.c** file, or the appropriate **.c** files reported during the model generate phase. For more information, refer to the User's Guide.

[4]Previously, **vsim.h** was **simutl.h**, which is what Intermetrics uses. The header files are different, therefore the filenames were changed to avoid confusion.

8. The `main()` routine has six subroutine calls. The four in the middle are deleted. These functions are either not supported by the current VHDL subset, or have been replaced by `init_cv()` and `sim_it()` above.

9. The name "`main()`" is changed to "`vhdl_main()`." With VSIM, the `main()` routine is found in the file `vsim.c`, which calls `vhdl_main()` if the simulation is run sequentially. If the simulation is run in parallel, the address of a `startup()` routine is passed as the starting address of each LP. From there, `startup()` calls `vhdl_main()` in the intermediate C code.

10. For getting output in the parallel environment, the name of each signal must be added to the signal structure after it is instantiated. For every `mksig()` function call, `mksig()` either returns a scalar or bit vector value, depending on the type of signal.

    - If `mksig()` is assigned to a variable such as `(*cd).Zxxxxxxx`, it is a scalar assignment. On the line below the mksig string, "`(*cd).PARM1 -> name = &(PARM2);`" is added where `PARM1` is the `Zxxxxxxx` string to which mksig is being assigned. `PARM2` is the first parameter that appears in the `m_signal` subroutine call that is six lines below.

    - If `mksig()` is not assigned to a scalar, then it's a bit vector assignment. Four lines above these assignments, the statement "`lastsig = sigarr + NUM1 - NUM2;`" is found, where `NUM1` and `NUM2` are integer values. Add "`loop_counter = NUM1 - NUM2;`" below that line. Then, after the line with the `mksig` string, the following statements are added:

```
temp_name = (char*)calloc(sizeof(PARM1) + 5, sizeof(char));
sprintf(temp_name, "%s(%d)", PARM1, loop_counter--);
(*(sigarr - 1)) -> name = temp_name;
```

      where PARM1 is a string which appear 7 lines below as `Z30000xxx.xxxxxxxxxxxx`.

11. Needless function calls are deleted. This was recommended—but not implemented—by Comeau since his data transformations were done by hand. Instead of deleting the calls, he wrote "dummy" functions. The following function calls are not required and are deleted:

    - `close_sigdict()`
    - `m_int_type()`
    - `m_real_type()`
    - `m_real_type()`
    - `m_signal()`
    - `pop()`
    - `push()`
    - `read_input()`
    - `rmtrrec()`
    - `rptstats()`
    - `rpterr()`
    - `Start_Nonarray_Comp()`
    - `sched()`
    - `timer()`
    - `tpop()`

12. Every behavior instance's "function behavior" is modified to report it's entity/architecture name if **MAPPING** is defined in VSIM and the boolean variable **mapping** is still true. Each of these function declarations is of the form **Zxxxxxxx_xxxx(bi)**. Inside the function, after local declarations, put the following:

```
#ifdef MAPPING
   if(mapping)
      printf("%s\n",  Zxxxxxxx_xxxx_trcbck);
#endif
```

    This step is also new, and an example is shown before and after in Appendix F.

```
ws              [ \t]*
comment         (\/\*)[^\n]*\n
include         #{ws}include[^\n]*\n
include_fn26    #{ws}include{ws}\"[Ff][Nn]26\"{ws}\n
trace           \{{ws}trace[^\n]*\n
if_trceqp       if{ws}\({ws}trceqp{ws}\){ws}
trc_or_trarec   [^\n]*((trace)|(TRAREC))[^\n]*\n
main            main{ws}\({ws}\){ws}[^;]
Z1_call         Z1([0-9]|[A-Z])*{ws}\({ws}\){ws};
Z5_function     Z5([0-9]|[A-Z])*{ws}\({ws}\){ws}[^;]
mksig_a         \(\*cd\)[^\n]*mksig
mksig_b         \n{ws}lastsig{ws}=
exec            \nZ([0-9]|[A-Z])*_[0-9]*{ws}\(bi\)
```

Figure 28. Regular Expressions Required to Identify Data to be Transformed

4.2.2 Lex Descriptions of the Transformation Steps. As each regular expression is matched in lex, the lex macros **ECHO**, **input()**, **output()**, and **unput()** are used in conjunction with a character stack to manipulate the source code according to the rules above. The **plex.l** file contains the lex description of these rules. Figure 28 shows the regular expressions and Figure 29 shows function calls used in the lex description to translate the data. These two Figures make up **plex.l**. For example, the definitions of Figure 28 show whitespace (**ws**) to be zero or more blank spaces or tabs; a comment is recognized by a ***** to the end of line (taking advantage of the intermediate C code's one-line comments); and an include directive is defined to be a pound sign, followed by white space, followed by the word **include**, to the end of line; etc. Then, the rules in Figure 29 use these definitions to recognize parts of the code that require modification and to implement those modifications.

The twelve steps of the postprocessor are accomplished in Figure 29 as follows:

1. *Step 1.* The function **check_include()** is called to remove the unnecessary directives.
2. *Step 2.* The **#include fn26** directives are deleted by not echoing them to the output file.

```
{comment}       { lineno++;           /* assumes comments on one line */
                  numcomments++;      /* count the comments */
                  ECHO;               /* echo comment to output */
                }
{include_fn26}  { lineno++;           /* do nothing, i.e., delete line */
                  output('\n');       /* output a newline */
                  num_inc_del++;      /* count deleted #include fn26 */
                }
{include}       { check_include();    /* evaluate and modify include directives */
                }
{trace}         { fix_trace();        /* {trace ... to {... */
                }
{if_trceqp}     { del_if_trceqp(); }  /* delete if(trceqp){...} structures */
{trc_or_trarec} { lineno++;           /* do nothing, i.e., delete line */
                  num_trc_or_trarec++;
                }
{main}          { do_main(); }        /* adjust main function */
{Z1_call}       { /* do nothing if in main, i.e., don't ECHO */
                  if (!found_main) ECHO;
                  else Z1_calls_del++;
                }
{Z5_function}   { do_Z5_function(); } /* modify Z5xxxx() functions */
{mksig_a}       { do_mksig_a(); }     /* modify bit mksig() function calls */
{mksig_b}       { do_mksig_b();       /* modify bit vector mksig calls */
                }
{exec}          { add_mapping(); }    /* add #ifdef MAPPING directive */
close_sigdict{ws}\(             { del_fn_call(); } /* delete function calls... */
m_int_type{ws}\(                { del_fn_call(); }
m_real_type{ws}\(               { del_fn_call(); }
m_signal{ws}\(                  { del_fn_call(); }
pop{ws}\(                       { del_fn_call(); }
push{ws}\(                      { del_fn_call(); }
read_input{ws}\(                { del_fn_call(); }
rmtrrec{ws}\(                   { del_fn_call(); }
rptstats{ws}\(                  { del_fn_call(); }
rpterr{ws}\(                    { del_fn_call(); }
Start_Nonarray_Comp{ws}\(       { del_fn_call(); }
sched{ws}\(                     { del_fn_call(); }
timer{ws}\(                     { del_fn_call(); }
tpop{ws}\(                      { del_fn_call(); }
\n              { lineno++;
                  ECHO;
                }
.               { ECHO; }
```

Figure 29. Function Calls and Actions Defined for Each Regular Expression

3. *Step 3.* The paths of all remaining include directives are modified in `check_include()`.
4. *Step 4.* `{trace...` is changed to `{...` by calling the `fix_trace()` function.
5. *Step 5.* Occurrences of `if(trceqp) {...}` are deleted in the `del_if_trceqp()` routine.
6. *Step 6.* Lines containing `trace` and `TRAREC` are deleted by not echoing them to the output file.
7. *Step 7.* The `Z5xxxxxx()` function is modified in `do_Z5_function()`.
8. *Steps 8 and 9.* The main function is modified by `do_main()`.
9. *Step 10.* Scalar signals are modified in `do_mksig_a()`, and bit vector signals are modified in `do_mksig_b()`.
10. *Step 11.* All unnecessary function calls are removed by calling `del_fn_call()`.
11. *Step 12.* Reporting of their entity/architecture name is added to behavior functions in `add_mapping()`.

Finally, when the intermediate C code has been completely transformed, a report is generated, such as shown in Figure 30.

4.3 Interfacing VSIM with SPECTRUM.

SPECTRUM provides support for running concurrent processes on the Intel iPSC/2 and iPSC/860 Hypercubes. For VSIM, the concurrent processes each run the VHDL simulation cycle as described in Chapter 3. The behaviors are partitioned among the processes and interprocess communication is accomplished via calls to SPECTRUM.

4.3.1 Main SPECTRUM Functions. All functions discussed in this section are listed in Appendix G.

4.3.1.1 Initialization. Prior to running a parallel simulation using SPECTRUM, the number of logical processes is specified in a header file. When the simulation begins, a call to `lp_level_init()` is made to establish the following:

- The LP relationships.
- The address of the starting procedure for each LP.

```
Approx lines:                              2710
Comments:                                     5
#include directives modified:                 5
#include directives removed:                13
{trace... changed to {...  :                28
if(trceqp) tests removed:                   35
"trace" or "TRAREC" lines removed:         223
Z1xxxxxx() calls removed:                     4
Z5xxxxxx() functions modified:                1
Scalar "mksig" assignments modified:         18
Bit vector "mksig" assignments modified:      0
#ifdef MAPPING added:                        14

Other function calls removed:

        close_sigdict():          1
        m_int_type():             0
        m_real_type():            1
        pop():                   21
        push():                  21
        read_input():             1
        rmtrrec():                0
        rptstats():               1
        rpterr():                23
        Start_Nonarray_Comp():    0
        sched():                  0
        timer():                  1
        tpop():                  31
```

Figure 30. Example Postprocessor Report

- The addresses of any filters used by all LPs.

LP relationships are specified by the user in a `lp.arcs` file. The specifications for this file are found in the SPECTRUM user's guide (16) and in the AFIT Parallel VHDL user's guide (4).

The function `vspec_init()` builds a table of function pointers for SPECTRUM. Each function pointer represents the starting code for the simulation on each LP. For VSIM, *all* LPs start with the routine `startup()`. Therefore, every entry in the array `functions[]` is loaded with the address of `startup()`. Finally, a call is made to SPECTRUM's `lp_level_init()`, where SPECTRUM initializes and each LP calls `startup()`. In turn, `startup()` calls the intermediate C codes `vhdl_main()`, where the circuit to be simulated is configured, and the simulation begins on each LP.

4.3.1.2 Sending Signal Changes. When VSIM identifies a signal change that is required by another LP, it uses a function calles `send_signal` to build an event and call SPECTRUM's `lp_post_event()`. SPECTRUM sends the event to the specified LP after the send filter performs the protocol-necessary functions, as discussed in the filter section.

4.3.1.3 Receiving Signal Changes. An LP receives a signal by making a call to SPECTRUM's `lp_get_event()`. The event is then made into a signal record and posted in the active list by a function called `receive_signal()`. If a null-pointer is returned from `lp_get_event()`, this indicates that no event was ready to return and the local LP can safely execute without an event from another LP. This determination is made by the receive filter.

4.3.1.4 Clock Management. VSIM and SPECTRUM each have local clocks for every LP—both implemented as an integer. When an LP updates the VSIM clock, it passes this time to SPECTRUM's `lp_advance_time()` to keep the clocks synchronized.[5]

[5]The SPECTRUM clock is synchronized with the VSIM clock in each LP. This does not mean that every LP has the same time—only that the VSIM clock and the SPECTRUM clock on *each* LP has the same time. The LPs run asynchronously by the rules of the null-message protocol.

4.3.2 Implementation of SPECTRUM Filters for VSIM. The filters are used to implement the null-message protocol for parallel simulations. The theory behind this protocol is discussed in Chapters 2 and 3. The filters used by VSIM are based on an existing set of filters called **chanclocks**, established at AFIT. The receive filter is modified for VSIM to take into consideration the two event queues—SPECTRUM's input event list and VSIM's active list.

Channel times are introduced to track the safe time and the output send times. As discussed in Chapter 3, safe time is defined as the minimum input channel time of all input arcs. Output channel times are tracked to avoid sending null messages when they are not necessary.[6]

4.3.2.1 The Initialization Filter. When VSIM calls `lp_init()`, an initialization filter is used to instantiate and initialize channel times for the input and output arcs defined in the `lp.arcs` file. Also, a null message is sent to every downstream LP with a time stamp of t_{LP_DELAY}.

4.3.2.2 The Send Filter. When an LP sends another LP a signal, `null_post_filter()` sends a null message to every other downstream LP with the same time stamp. Also, the channel time for each output arc is updated.

4.3.2.3 The Receive Filter. The receive filter, `null_get_fltr()`, is used to get events from upstream LPs. It determines if the local LP is able to prooceed, i.e., at least one message has been received from each upstream LP and the time of the next event in SPECTRUM's queue is less than the safe time. If so, the event is valid (no event will be received with an earlier time stamp) and returned to VSIM.[7]

If the LP cannot return a message, it "peeks" at VSIM's active list to get the next event time. If this time is less than the safe time, the filter returns, causing a NULL pointer to be returned

[6] Since null messages are only used to avoid deadlock, if a message has been sent to another LP at time t, there is no need to (possibly) send another null message to the same LP at time t.

[7] Input null messages are stripped out.

by `lp_get_event()`. When VSIM receives the NULL pointer, it proceeds without adding a new record to the local active list.

If the receive filter cannot return a valid message and the next event time is greater than the safe time, then the filter blocks after sending a null message to every downstream LP guaranteeing a message is not sent any sooner. In this way, deadlock is avoided. The rule, as discussed in Chapter 3, is an LP sends a null message to every downstream LP with a time stamp equal to either VSIM's next event time or the sending LP's safe time plus output delay.

4.3.3 Termination. When an LP has completed the simulation, it builds a null message with the maximum simulation time and sends it to all downstream nodes. Then it calls `node_terminate`, which signals to the host that the LP's simulation has completed. Ideally, a terminate filter should be used instead of relying on the application to create and send a null message *and* make a node-level function call.[8]

[8]Such a filter now exists in the latest version of SPECTRUM. VSIM uses this new version, but it does not use a terminate filter. Modification should be relatively straightforward and simple.

V. Results

5.1 Introduction.

In this chapter, the performance of several VHDL circuit simulations is discussed. First, three small adders are presented: An 8-bit carry save adder, an 8-bit carry propagate adder, and an 8-bit carry lookahead adder. Then, two larger circuits are simulated: A 16-bit bit/byte shifter and an 8 x 8 Wallace Tree multiplier with a 16-bit product.

With the exception of the 16-bit shifter, each circuit is compiled and run on both the iPSC/860 and the iPSC/2. Data is presented separately. The shifter produces a C code representation of the configuration file that is too large to compile on the iPSC/2; therefore, only iPSC/860 data is presented for the shifter. The largest circuit in terms of numbers of gates—the Wallace Tree—*did* compile on the iPSC/2 due to the hierarchical circuit design and use of incremental configurations.

All one-LP simulations represent the entire circuit as a single process on one node. One-LP simulations are the baseline for speedup calculations.

For performance measurements, each configuration is run 30 times and averaged. The total time for one simulation is considered to be the maximum time of all concurrent processes. Unless otherwise noted, all output is turned off and 20 input vectors or sets of vectors are applied to each circuit, e.g., 20 pairs of vectors are applied to the multiplier, 20 vectors are applied to the shifter, etc.

5.2 Program Validation.

Programs are validated by comparing them with Intermetrics' output. The process is as follows:

1. Run the simulation using Intermetrics' simulator.

```
TIME  |----------------------------SIGNAL NAMES----------------------------|
      |
(NS)  |    CIN    X(7 DOWNTO 0)    Y(7 DOWNTO 0)    COUT    Z(7 DOWNTO 0)
      |
   0  |    '0'     "00000000"       "00000000"      '0'      "00000000"
  10  |            "01010101"       "01001100"
  16  |                                                      "00011001"
  21  |                                                      "10010001"
  24  |                                                      "10000001"
  30  |    '1'     "10101010"       "10110011"              "10100001"
```

Figure 31. Sample Intermetrics Output for Carry Lookahead Adder

2. Generate an Intermetrics report. Example output for a portion of the carry lookahead adder simulation is shown in Figure 31. The circuit adds two 8-bit vectors, X and Y, along with a carry in, CIN (see the schematic on page 66). Figure 31 shows the values 01010101 and 01001100 are applied to the adder though X and Y respectively, while CIN remains a zero. The sum, Z, is 10100001 at 30 ns; and the carry output, COUT, remains a zero. Also, X, Y, and CIN are given new values to begin another addition (whose result is not shown).

3. After filtering the intermediate C code through the postprocessor and linking with VSIM, run the simulation in sequential mode under VSIM. An example of this output for the same portion of the carry lookahead adder is shown in Figure 32. Note the output of VSIM shows only the bits that have changed in each bit vector. For example, at 30 ns only bit 5 of Z has changed (from a zero to a one). This output can be directly mapped to the output of Figure 31.

4. Sort the output from the VSIM sequential run by time and signal name, respectively.[1]

5. Validate this output by comparing with the Intermetrics report.

[1]The output is already in time order; however, this sort organizes the signals while maintaining the time order.

```
10 ns, X(0) from 0 to 1
10 ns, X(2) from 0 to 1
10 ns, X(4) from 0 to 1
10 ns, X(6) from 0 to 1
10 ns, Y(2) from 0 to 1
10 ns, Y(3) from 0 to 1
10 ns, Y(6) from 0 to 1
16 ns, Z(0) from 0 to 1
16 ns, Z(3) from 0 to 1
16 ns, Z(4) from 0 to 1
21 ns, Z(3) from 1 to 0
21 ns, Z(7) from 0 to 1
30 ns, CIN from 0 to 1
30 ns, X(0) from 1 to 0
30 ns, X(1) from 0 to 1
30 ns, X(2) from 1 to 0
30 ns, X(3) from 0 to 1
30 ns, X(4) from 1 to 0
30 ns, X(5) from 0 to 1
30 ns, X(6) from 1 to 0
30 ns, X(7) from 0 to 1
30 ns, Y(0) from 0 to 1
30 ns, Y(1) from 0 to 1
30 ns, Y(2) from 1 to 0
30 ns, Y(3) from 1 to 0
30 ns, Y(4) from 0 to 1
30 ns, Y(5) from 0 to 1
30 ns, Y(6) from 1 to 0
30 ns, Y(7) from 0 to 1
30 ns, Z(5) from 0 to 1
```

Figure 32. Sample VSIM Output for Carry Lookahead Adder

6. Run the simulation in any parallel configuration, concatenate the LP output files, sort them by time and signal name, and use `diff` to compare them with the validated output.

Input test vectors for the adders are taken from Comeau (10:5-15). His test vector patterns are designed to verify the individual logic gates each act correctly for all possible inputs. For the shifter, patterns are chosen to verify that logic 1s and 0s shift left or right one or eight bits, depending on the input control signals. Also, 0s shifted in (one or eight bits) are verified. The multiplier is tested to verify limits and various intermediate values. For example, input pairs (0, 0), (0, number), (number, 0), (0, max), (max, 0), (number, max), and (max, number), and several combinations of (number, number) are tested and verified.

5.3 Circuit Partitioning.

No attempt is made to find the *optimal* circuit partitions; however, the absence or presence of speedup is discussed for each simulation. In general, larger or more complex simulations exhibit better speedup. Even though the presence of feedback can significantly inhibit performance in the null message protocol, very large circuits can still achieve speedup through parallel simulation.

The full adders that make up the carry save and carry propagate adders are partitioned symmetrically. For eight-LP simulations, each full adder is assigned to an LP, for four-LP simulations, two full adders are assigned to each LP, etc.

The partitioning for the carry lookahead adder is from Comeau's research. This adder is partitioned to avoid imposing feedback among LPs and to reduce the number of behaviors on successive downstream LPs (10:5-2).

Due to the large number of behaviors, the 16-bit shifter and the multiplier are simulated with a uniform random distribution of behaviors to LPs. Even though these circuits are combinational and "feedforward," such a distribution imposes feedback among the LPs. The results of this research

indicate that these larger circuits can be correctly simulated; therefore, more aggressive partitioning strategies can be investigated in the future.

5.4 Explanation of Charts.

For each chart, the performance of the ***simulation time*** and the ***total LP time*** are presented. The difference is that the simulation time represents the total time for an LP to execute the core simulation algorithm, as presented in Figure 21 on page 36. The LP time represents the total time an LP executes, i.e., overhead is included for SPECTRUM initialization, behavior and signal instantiation, and close-out.

All data is summarized in Appendix E.

5.5 Circuit Simulations.

5.5.1 Carry Save Adder. The 8-bit carry save adder, shown in Figure 33, is composed of eight independent full adders. The simulation has a total of 64 behaviors. Circuit partitioning is straightforward due to the lack of communication among the full adders.

Figure 34 shows the performance and speedup of the carry save adder for the iPSC/2. Note that the simulation loop exhibits ***superlinear speedup***, i.e., speedup increases greater than the number of LPs. This is due to the significantly reduced search and post times in each active list, as well as the reduced number of behaviors executing on each LP.

In parallel simulations, each LP maintains an active list that contains signals that only affect behaviors belonging to that LP. The total number of behavior executions, and therefore the total number of signal records generated, is dynamic. If there are m behaviors and n signal records posted in one circuit simulation, a sequential simulation may be bound by $O(n^2m)$ since each signal change

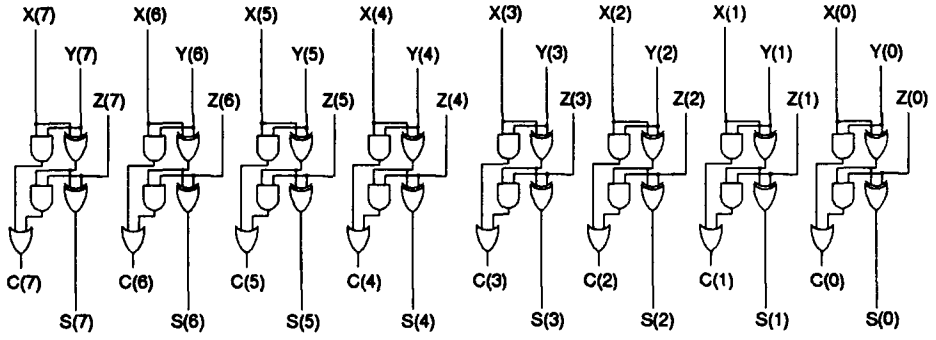


Figure 33. Schematic Diagram of the 8-bit Carry Save Adder (10)

(n) corresponds to up to m behaviors executing[2], which then posts the resulting signal into the active record list in $O(n)$ time.

If this simulation is now divided evenly between two LPs that require no communication between them (trivially parallel), then one LP now has $m' = m/2$ behaviors, and the total number of signal records generated and posted can be estimated to be $n' = n/2$. The overall execution is then $O((n')^2/m') = O(n^2m/8)$. This means that the execution time for a ***trivially parallel*** circuit evenly distributed between two nodes can execute as much as eight times faster as a sequential simulation of the same circuit. Likewise, trivially parallel, balanced circuits partitioned among four and eight nodes can execute 64 and 512 times faster, respectively. This, of course, is a very high bound on speedup estimations because the number of generated signals is estimated, and the

[2]This corresponds to one time through the simulation loop. This is a very high estimation, as one signal change rarely directly affects every component of a circuit.

number of behaviors scheduled for execution due to one signal change is almost always significantly less than the total number of behaviors.

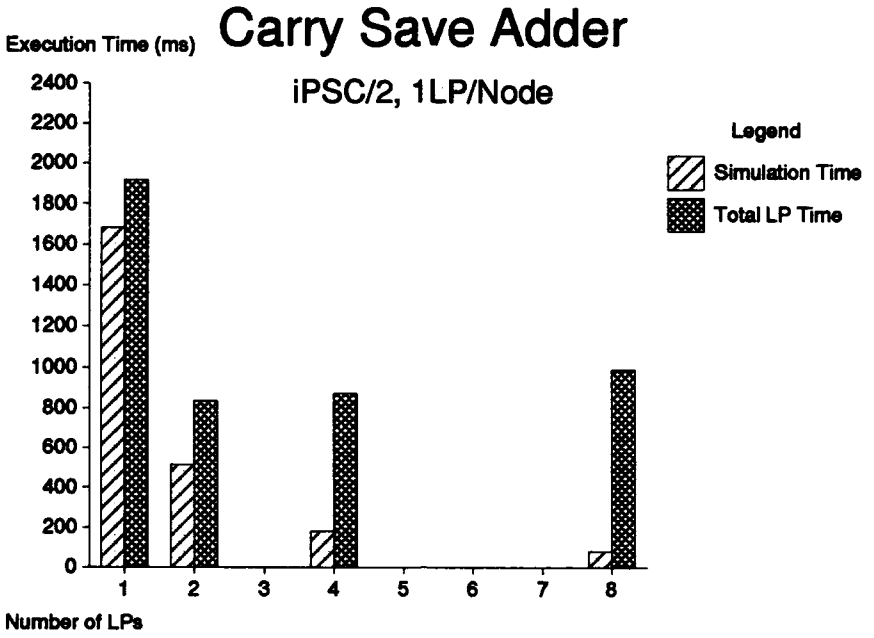
Also Figure 34 shows the overall LP time for the carry save adder was never below $800ms$. Therefore, although each simulation loop was improving in performance, the overall LP time is bounded by SPECTRUM's initialization and close-out functions. As other simulations show, this limitation disappears as circuit sizes and complexities increase.

For the iPSC/860, Figure 35 shows the computation times are *significantly* reduced as the number of LPs is increased. Also, the total execution time is less for SPECTRUM overhead, however, it increases with number of LPs due to increased contention for common resources, like input files and node-to-host synchronization.

5.5.2 Carry Propagate Adder. With the 8-bit carry propagate adder of Figure 36, the carry output of each adder is "propagated" to the next full adder. This introduces communication among the LPs. Otherwise, partitioning is the same as that of the carry save adder. Adjacent full adders are assigned to the same LP in order to reduce LP communications. The simulation of the carry propagate adder consists of 57 behaviors.

For the iPSC/2, Figure 37 shows a maximum simulation-loop speedup of about 2.3 for either two or four LP configurations. The total LP time shows a speedup of about 1.5 for four LPs. At eight LPs, the communications overhead overcomes the computation, and no speedup is obtained. For the iPSC/860 simulations of Figure 38, the simulation time shows a modest 1.2 speedup on two LPs; however, the overall LP time shows no speedup whatsoever. As is the case with the carry save adder, the carry propagate adder is too small to show much promise of speedup on the iPSC/860—regardless of the addition of LP communication requirements.[3]

[3] "Too small" can mean either a small number of components (behaviors), or a small number of test vectors, since each contributes to greater active lists and numbers of behaviors scheduled. Therefore, if the number of input vectors (test vectors) were increased sufficiently, the same carry propagate adder may no longer be "too small."



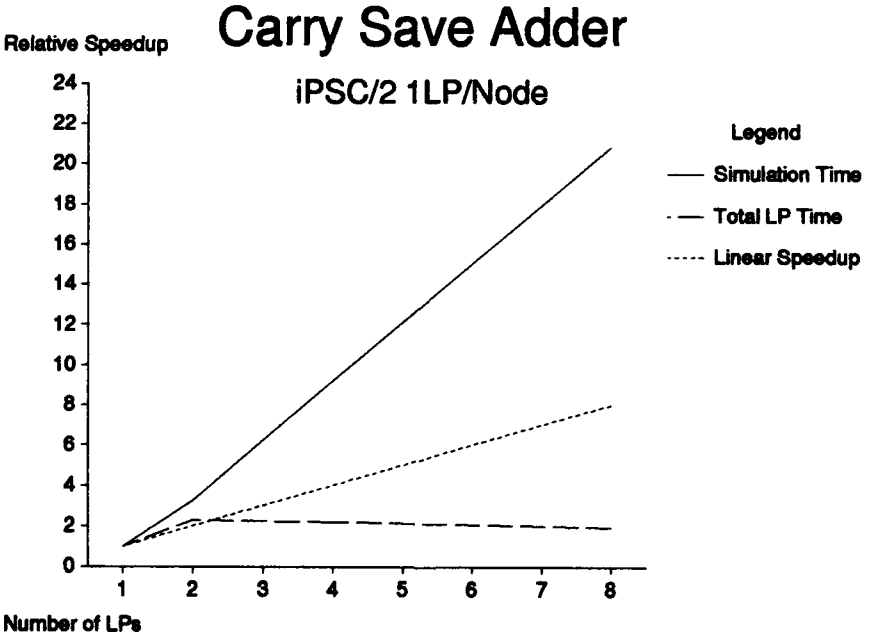


Figure 34. Performance of the Carry Save Adder on the iPSC/2

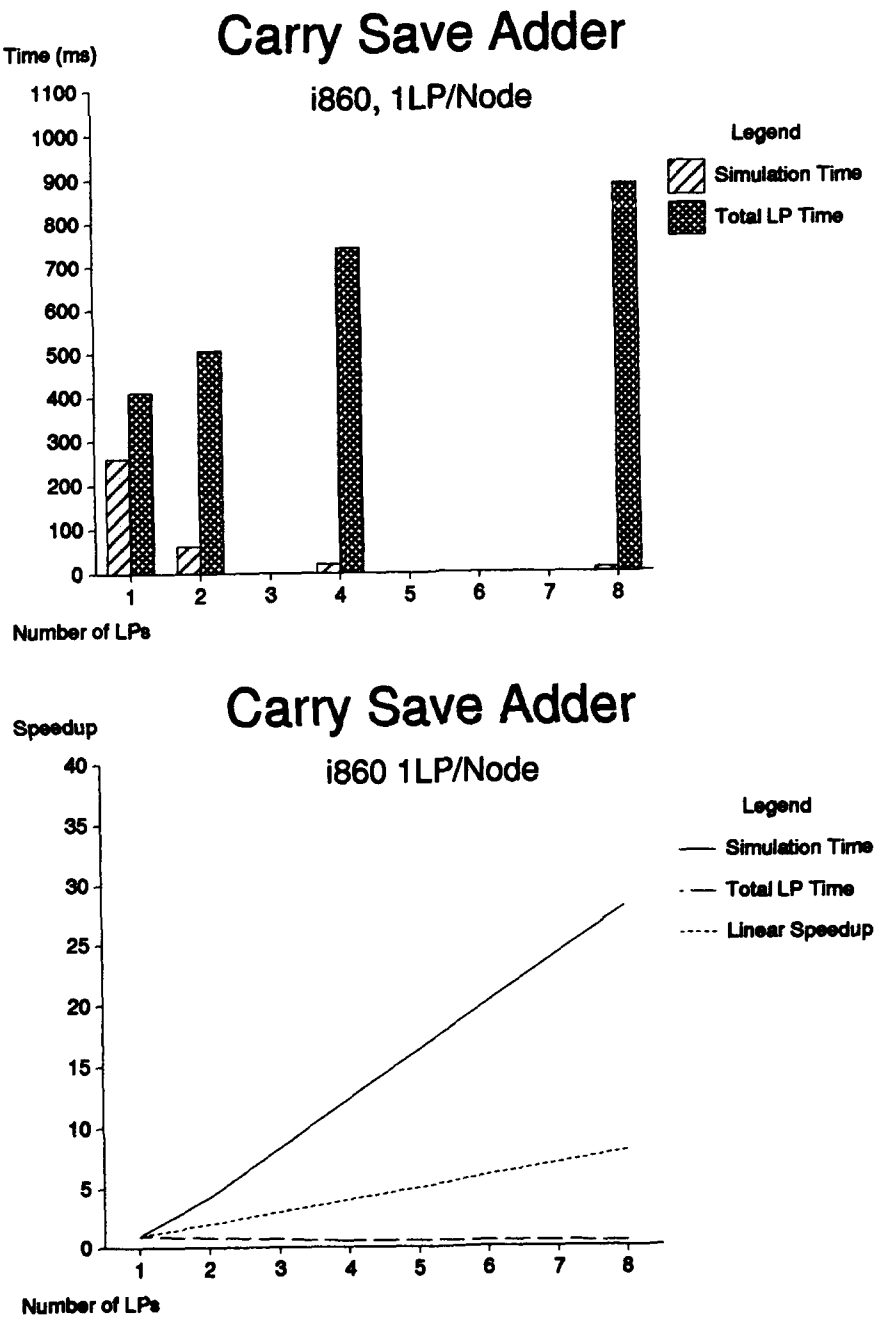


Figure 35. Performance of the Carry Save Adder on the iPSC/860

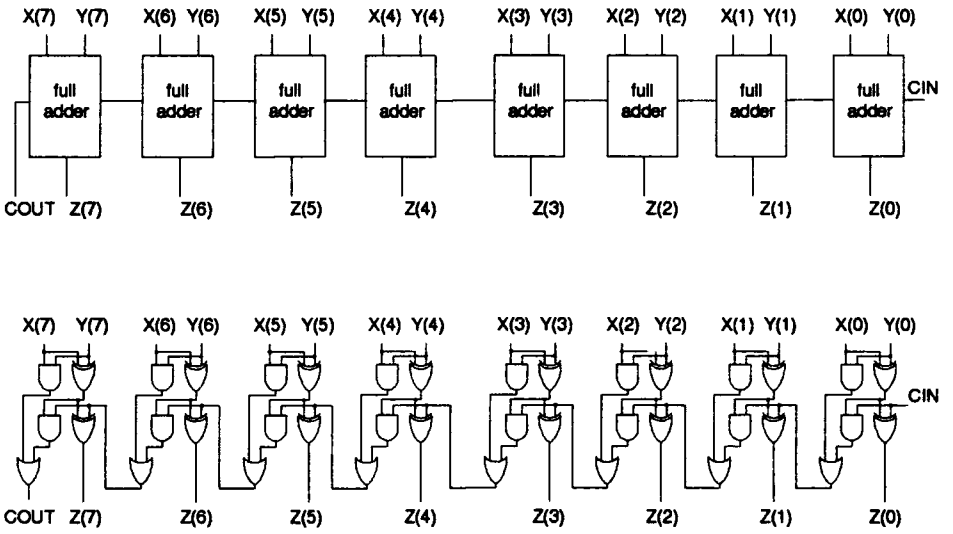


Figure 36. Schematic Diagram of the 8-bit Carry Propagate Adder (10)

5.5.3 Carry Lookahead Adder. The 8-bit carry lookahead adder is made of two 4-bit carry lookahead adders, as shown in Figure 39. The simulation consists of 77 behaviors. For two-LP simulations, each 4-bit adder is assigned an LP. Four- and eight-LP simulations are partitioned to avoid imposing feedback, as well as to "front load" upstream LPs with more behaviors, as shown in Figures 40 and 41. Partitioning is shown for only the lower 4-bit adder; however it is the same for the upper 4-bit adder.

For carry lookahead adder simulations on the iPSC/2, shown in Figure 42, all multi-LP simulations exhibited speedup over the one-LP simulation. The best speedup for this circuit is 2.5, which occurs for the four-LP simulations. This circuit is "larger" than the two previous adders, and the overall LP time more closely follows the trends of the "inner" simulation times. As circuits continue to grow, this becomes more and more apparent. On the iPSC/860, however, the computation time of the node processors still overcomes the benefits of partitioning the circuit, as shown in Figure 43.

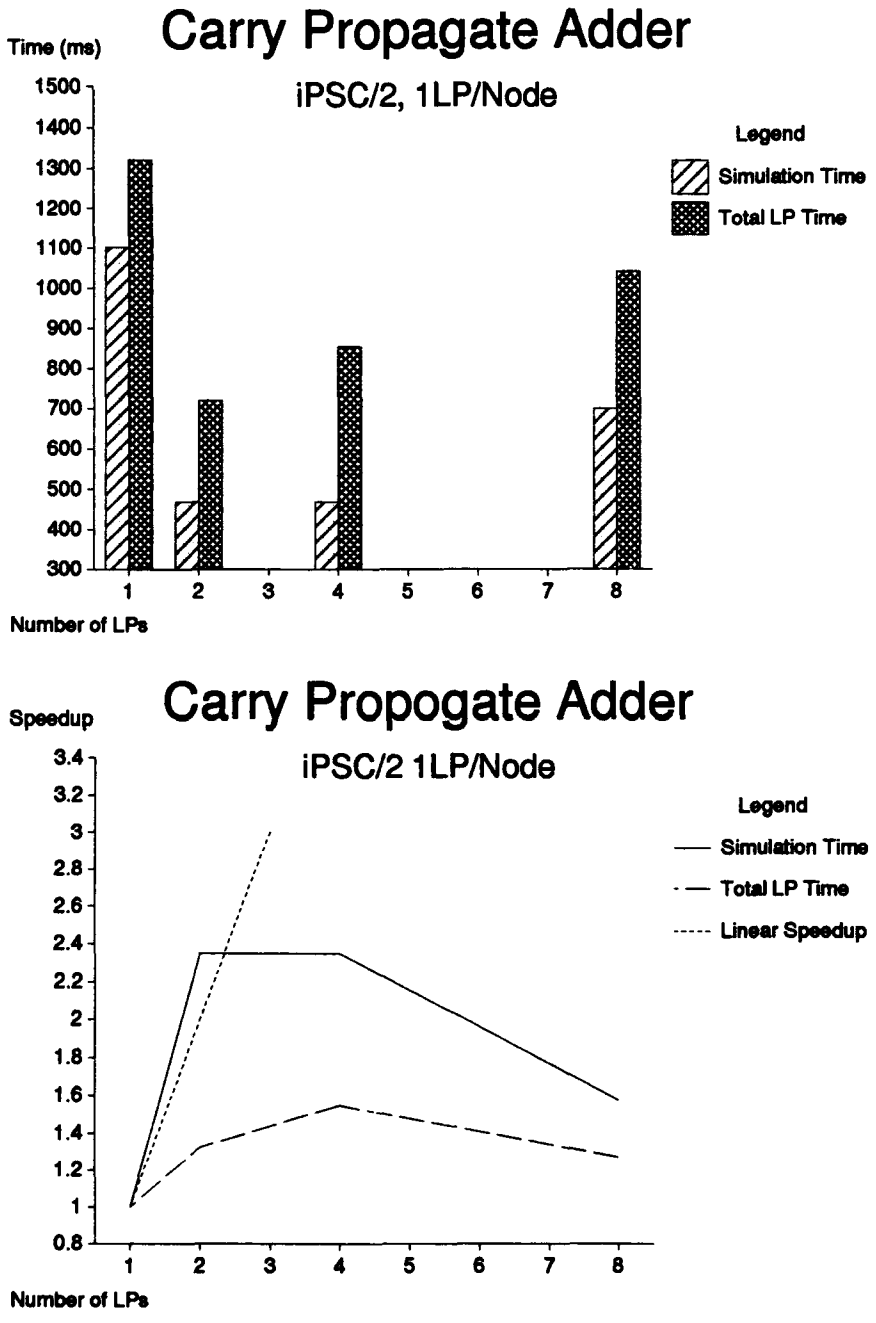
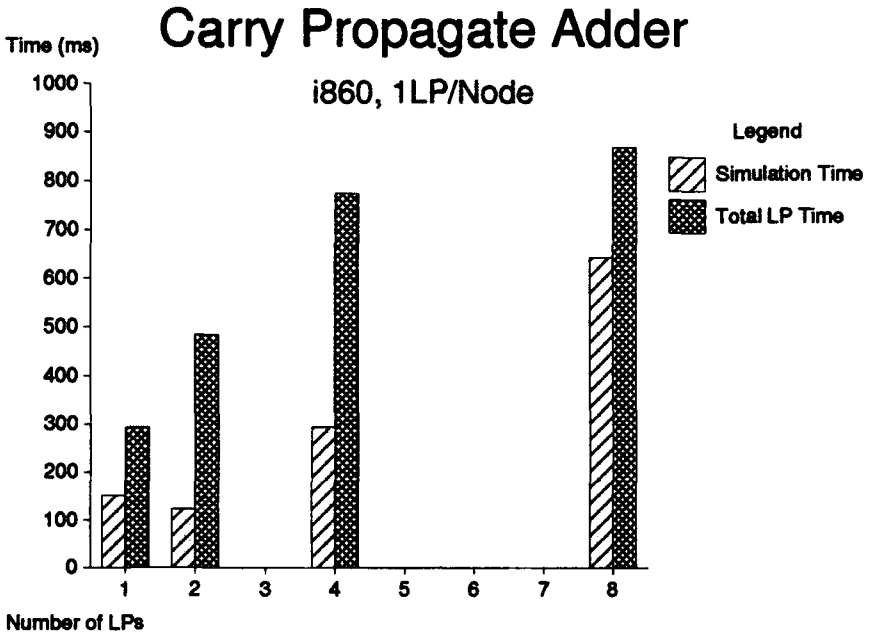


Figure 37. Performance of the Carry Propagate Adder on the iPSC/2



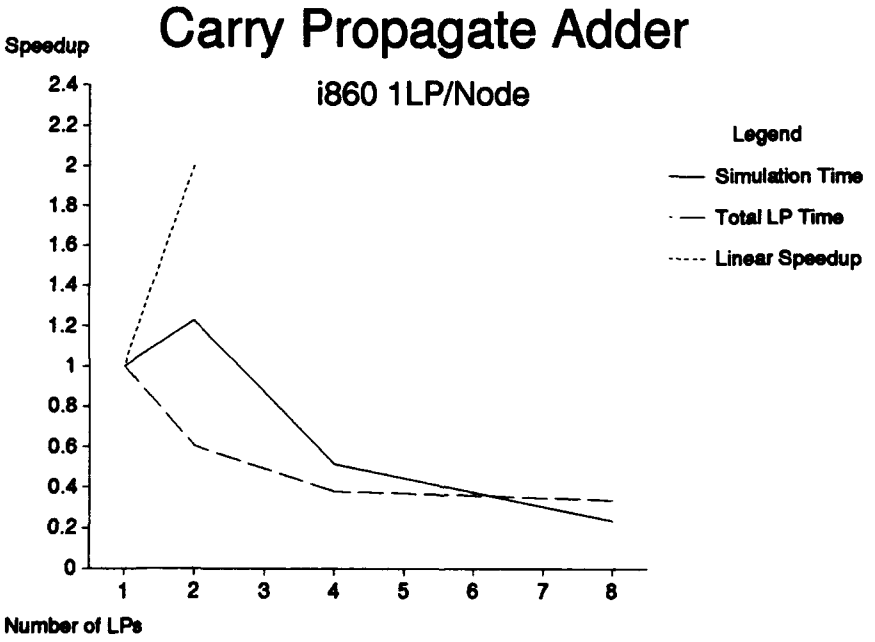


Figure 38. Performance of the Carry Propagate Adder on the iPSC/860

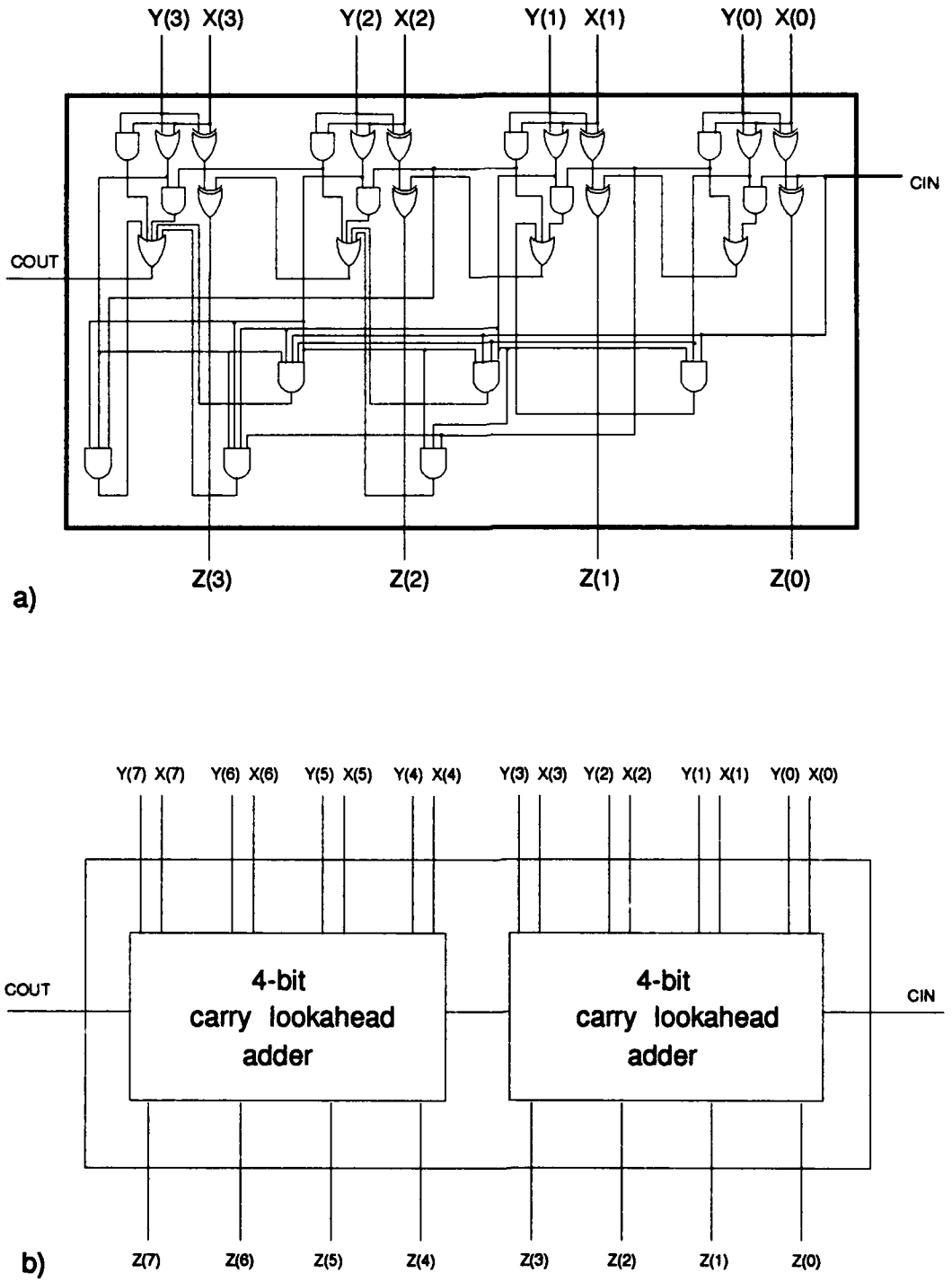


Figure 39. Schematic Diagram of the 8-bit Carry Lookahead Adder (10)

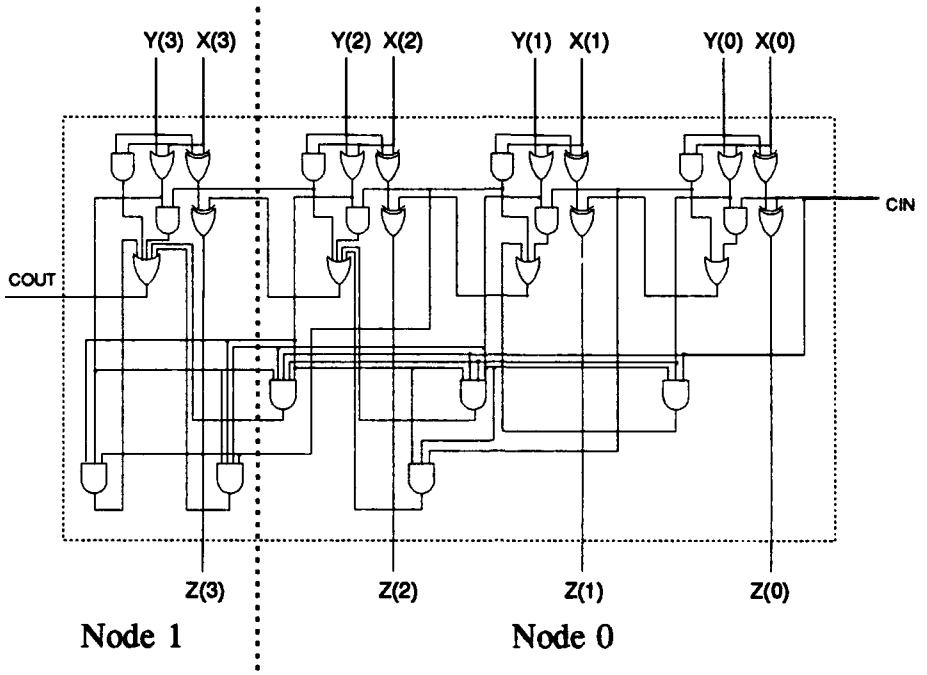


Figure 40. Four-LP Partition of the Carry Lookahead Adder (Lower Four Bits Shown)

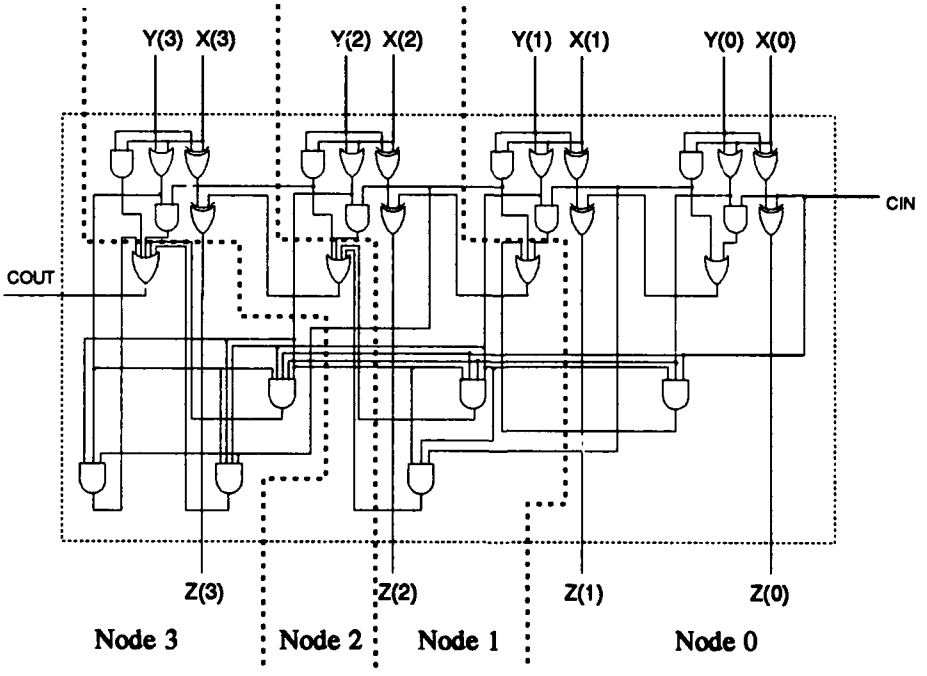
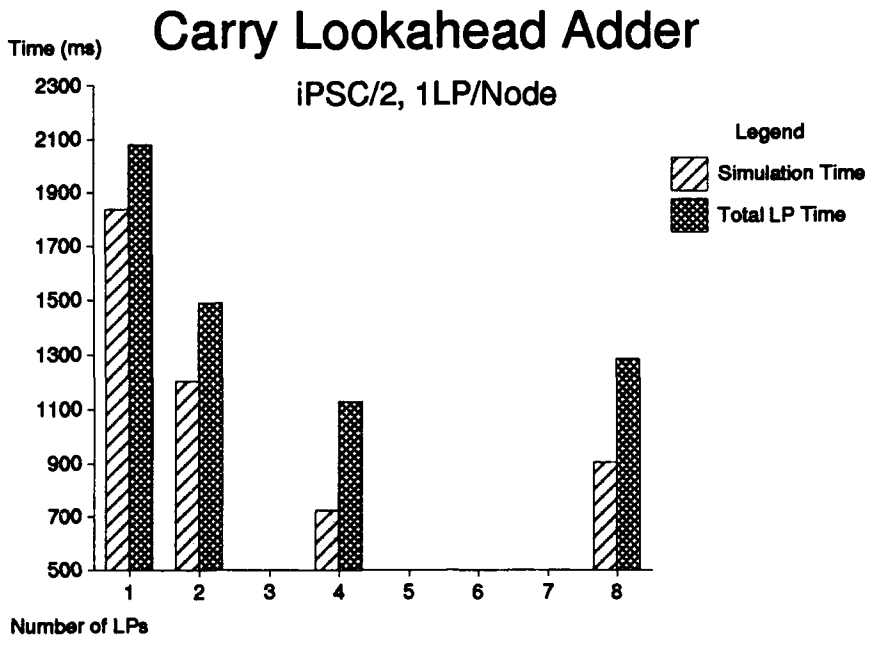


Figure 41. Eight-LP Partition of the Carry Lookahead Adder (lower Four Bits Shown)



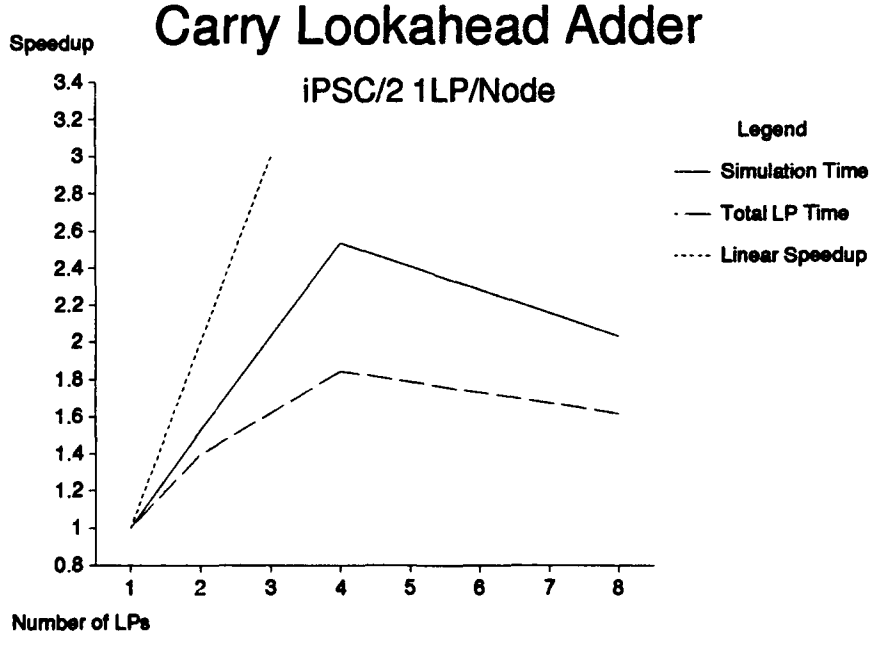
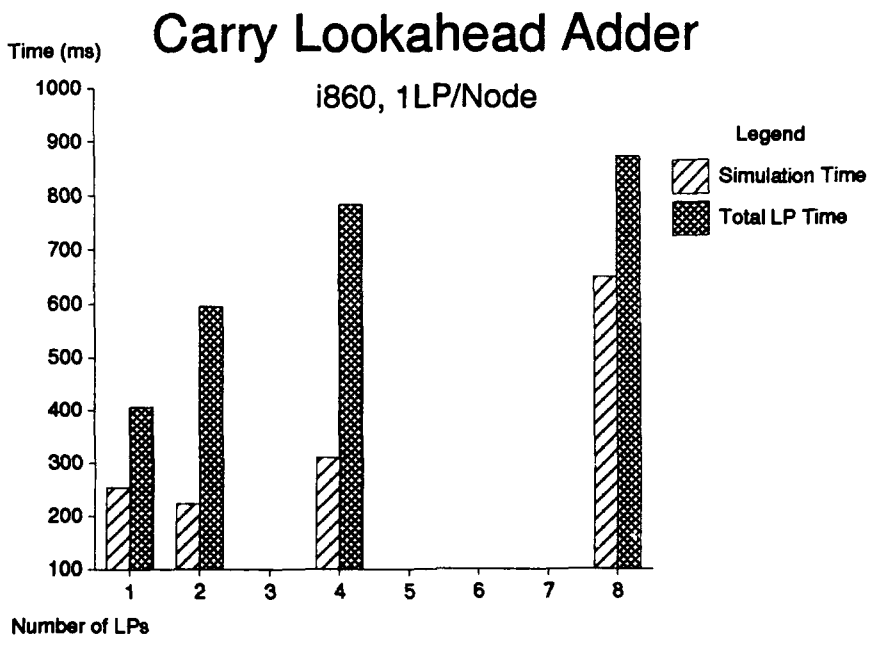


Figure 42. Performance of the Carry Lookahead Adder on the iPSC/2



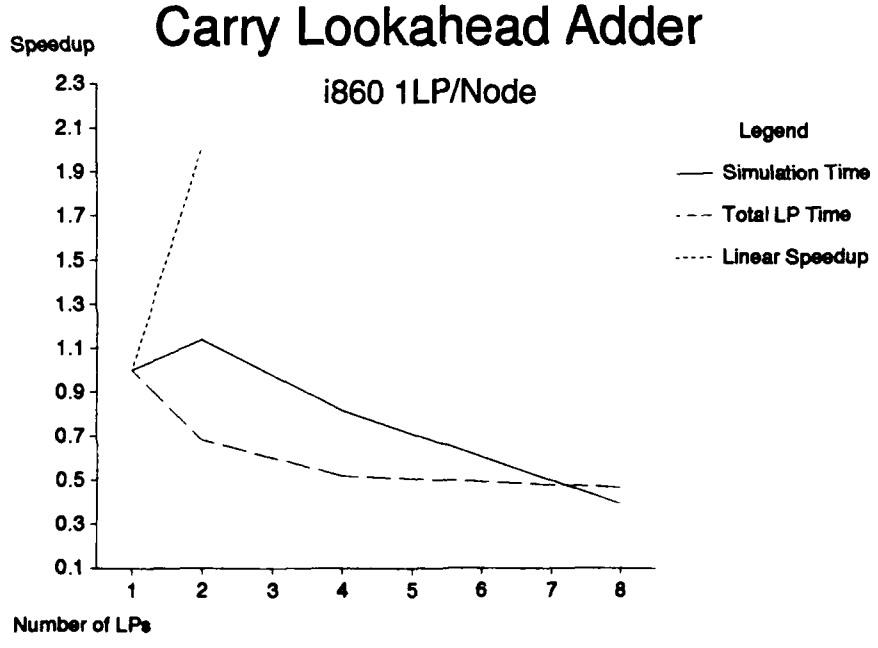


Figure 43. Performance of the Carry Lookahead Adder on the iPSC/860

5.5.4 Shifter. The 16-bit shifter shifts a 16-bit word either one or eight bits right or left depending on the control inputs. The schematic diagram is shown in Figure 44. Simulation of this shifter contains 309 behaviors. Therefore, partitioning is accomplished via a uniform random distribution of behaviors-to-LPs. This means feedback is artificially introduced due to LP communication. Therefore, performance in parallel configurations is not very promising, as shown in Figure 45. It does demonstrate that larger simulations can be correctly simulated in parallel on the iPSC/860.

5.5.5 Multiplier. The Wallace Tree Multiplier is the largest circuit tested. It contains 1050 behaviors. Schematic diagrams and a description of the hierarchical design are included in Appendix D. Behavior partitioning is once again random. Fortunately, the multiplier was created with a hierarchical set of components and configuration descriptions, and the corresponding C code is not too large for either the iPSC/2 or the iPSC/860.

Figures 46 and 47 show multiplier performance on the iPSC/2 and iPSC/860, respectively. Both hypercubes demonstrate increasing speedup as the circuit is simulated on two and then four LPs. This is encouraging and somewhat surprising since the random partitioning again imposes feedback among LPs. Because of these results, greater performance improvements can be expected for very large circuits if partitioning algorithms can be generated to avoid excess LP feedback.

5.6 Performance vs. Test Vector Quantity.

The carry lookahead adder is now modified to apply 64 pairs of input vectors instead of 20. This corresponds to a larger initial active list, more active records, and therefore more executions of behaviors. Figure 48 shows the corresponding speedup increases as the number of LPs increase. The maximum speedup here is 6.69 for eight LPs. For the iPSC/860 of Figure 49, speedup also improves, but the maximum is 3.75 for four LPs. These trends were similar for the other circuits.

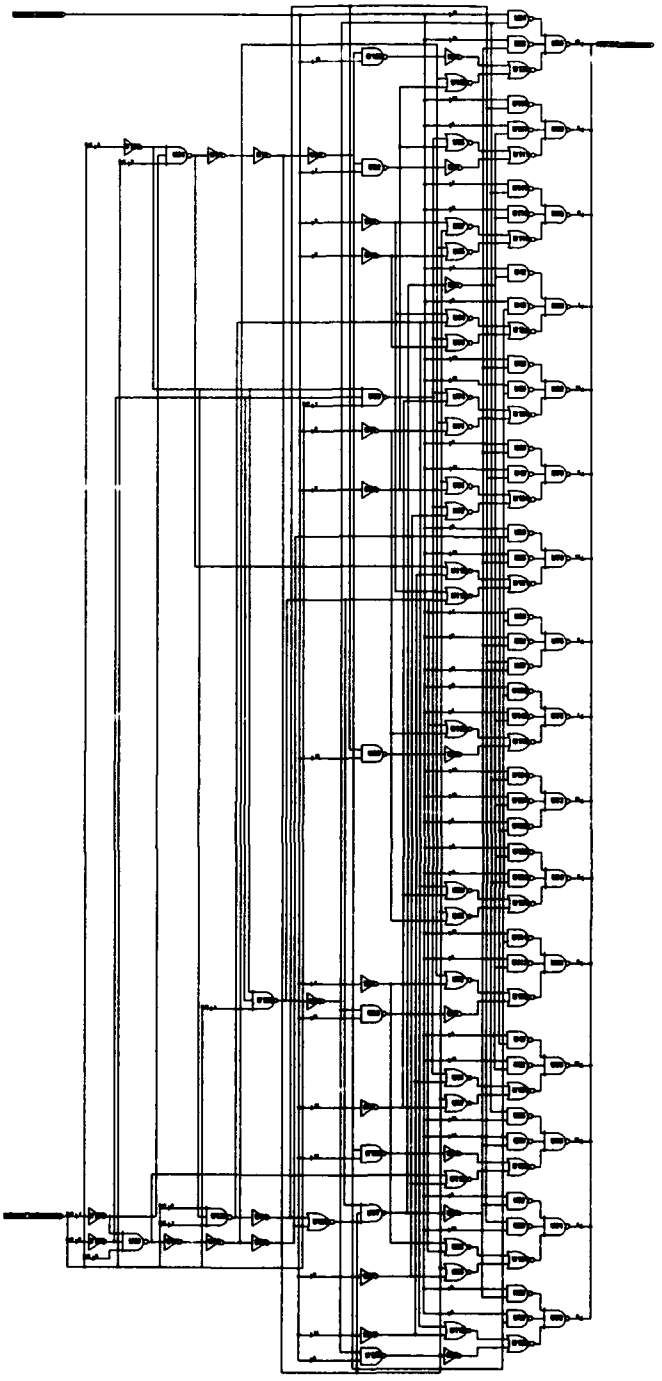


Figure 44. Schematic diagram of the 16-bit shifter

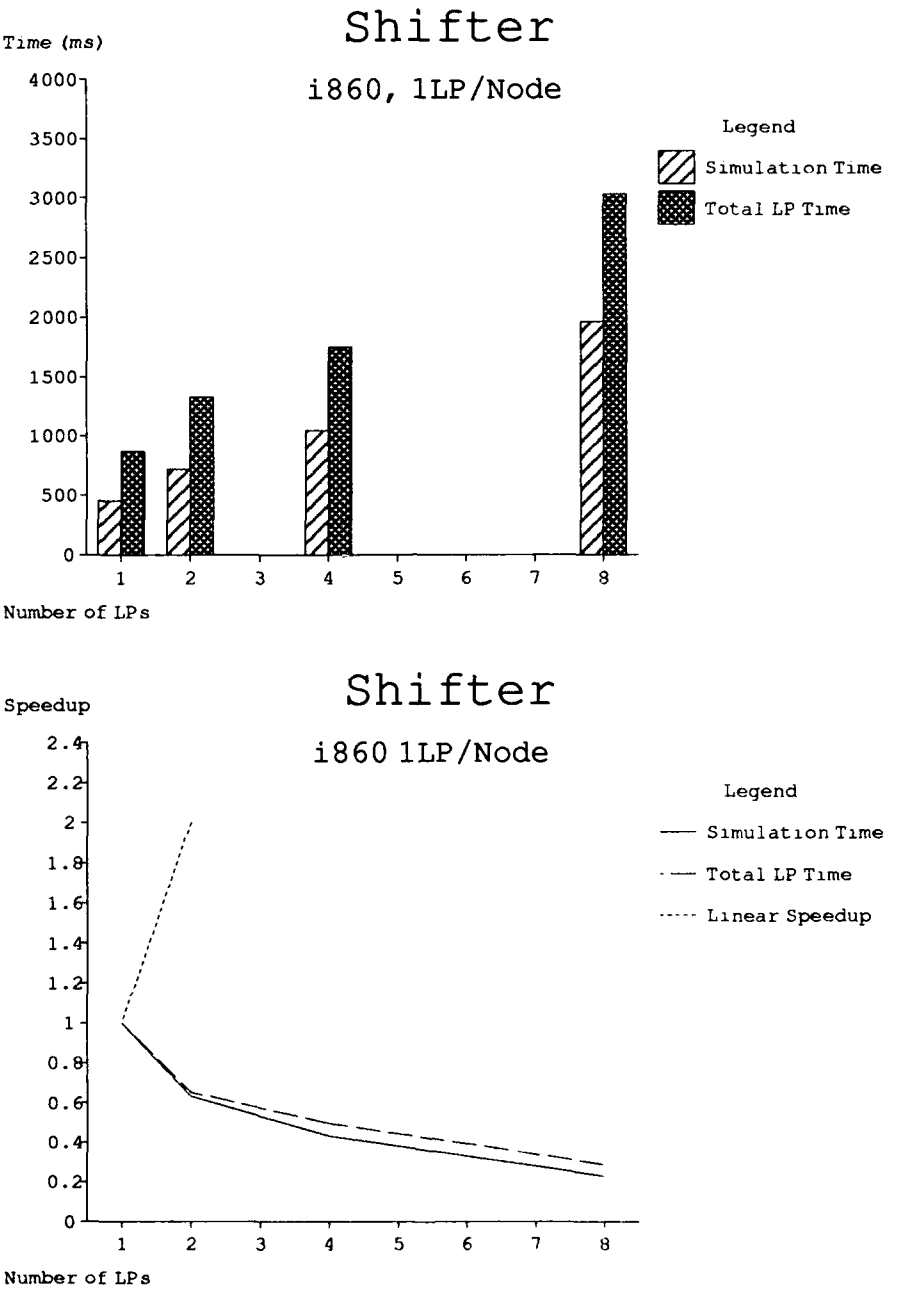
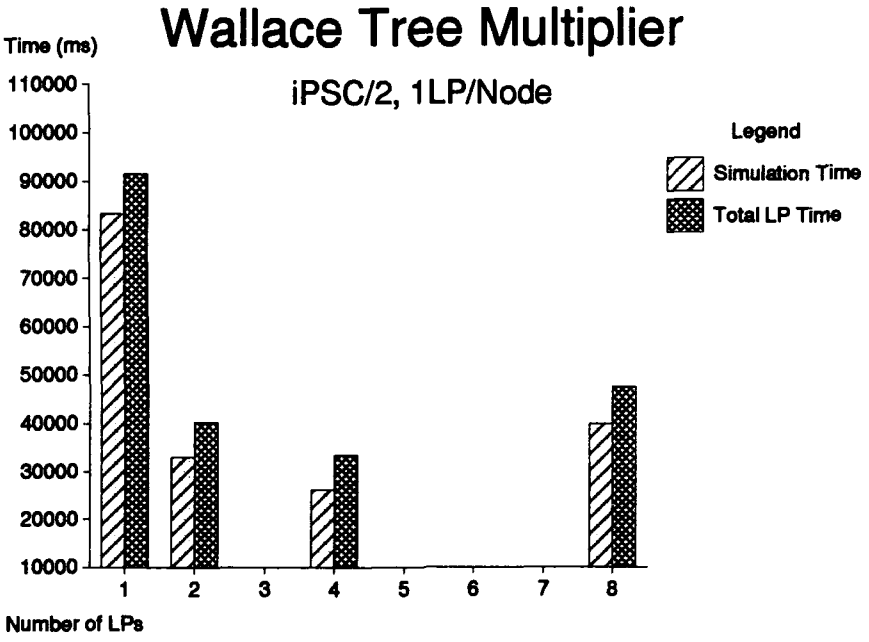


Figure 45. Performance of the 16-bit Shifter on the iPSC/860



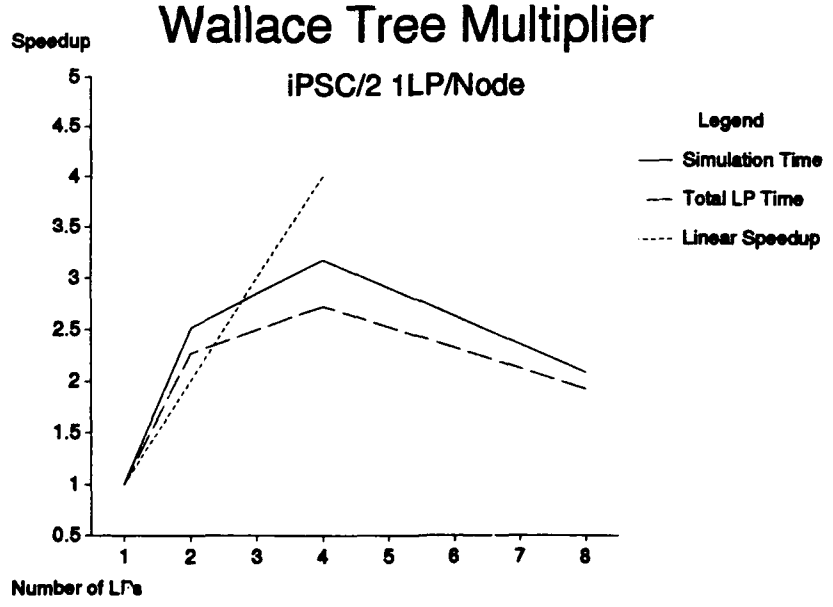


Figure 46. Performance of the Wallace Tree Multiplier on the iPSC/2

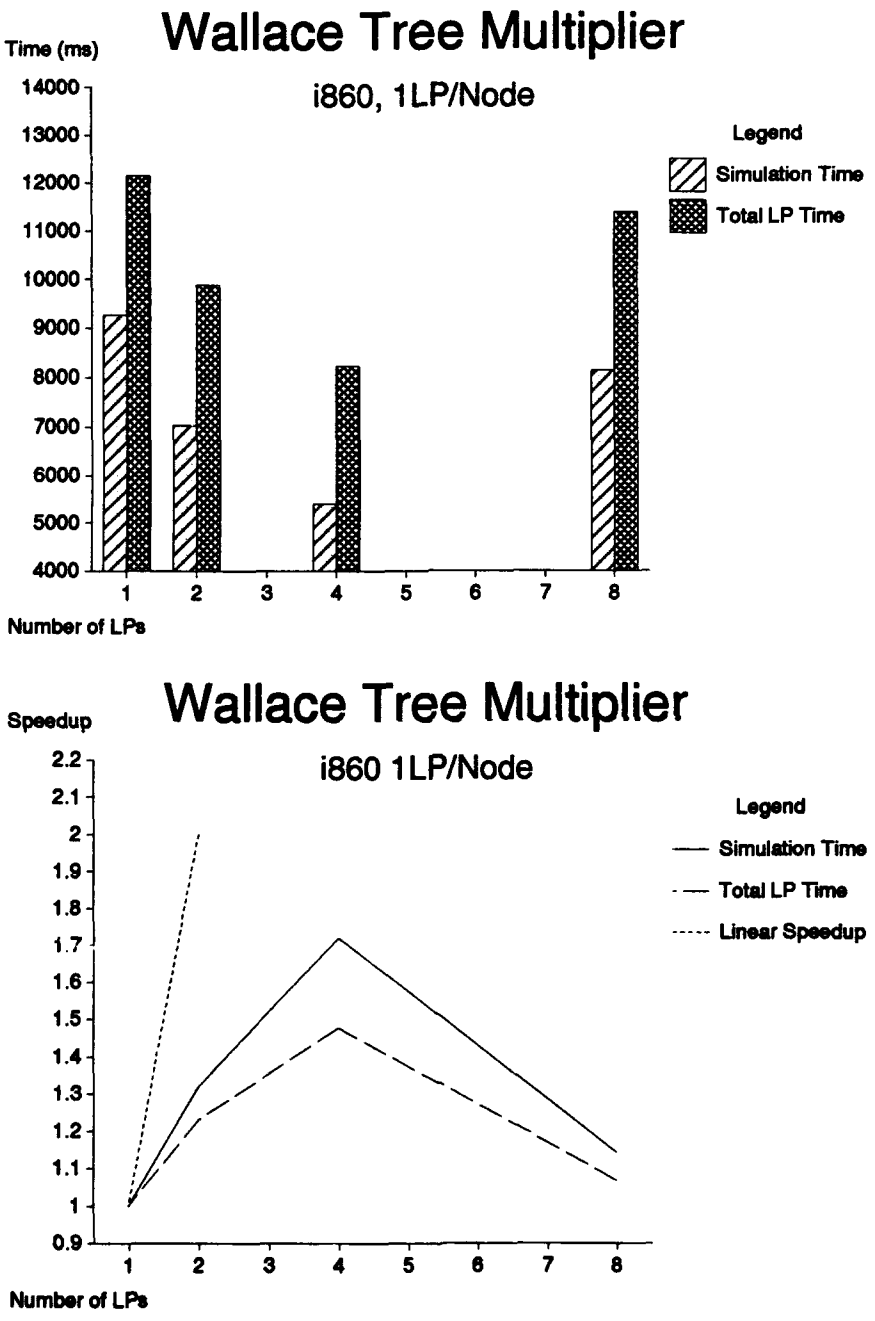
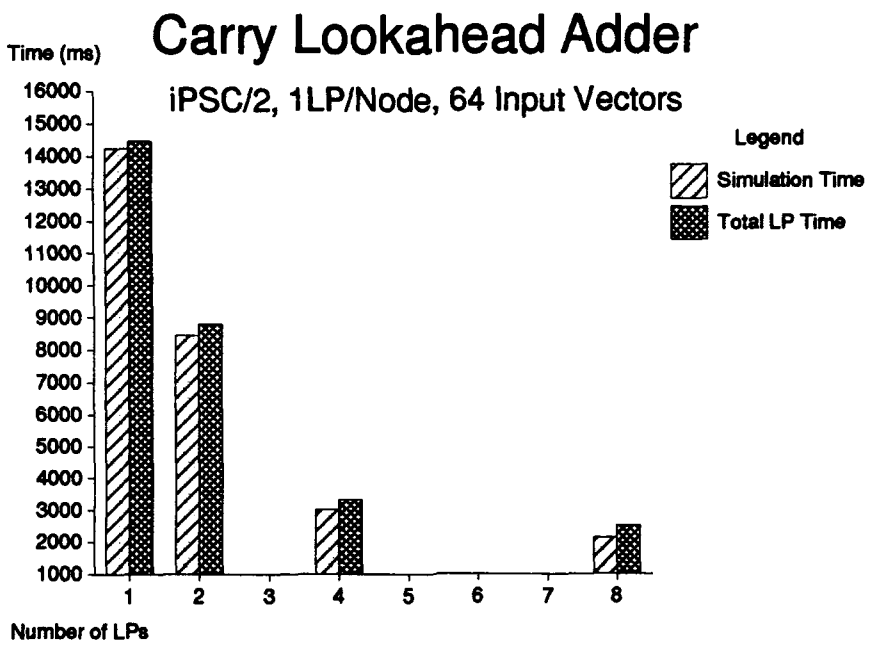


Figure 47. Performance of the Wallace Tree Multiplier on the iPSC/860



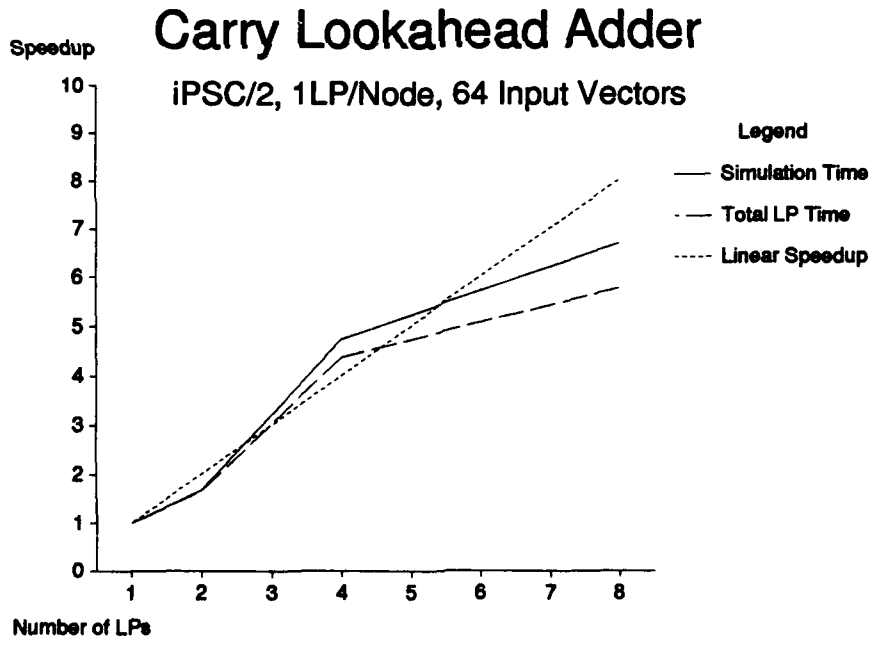
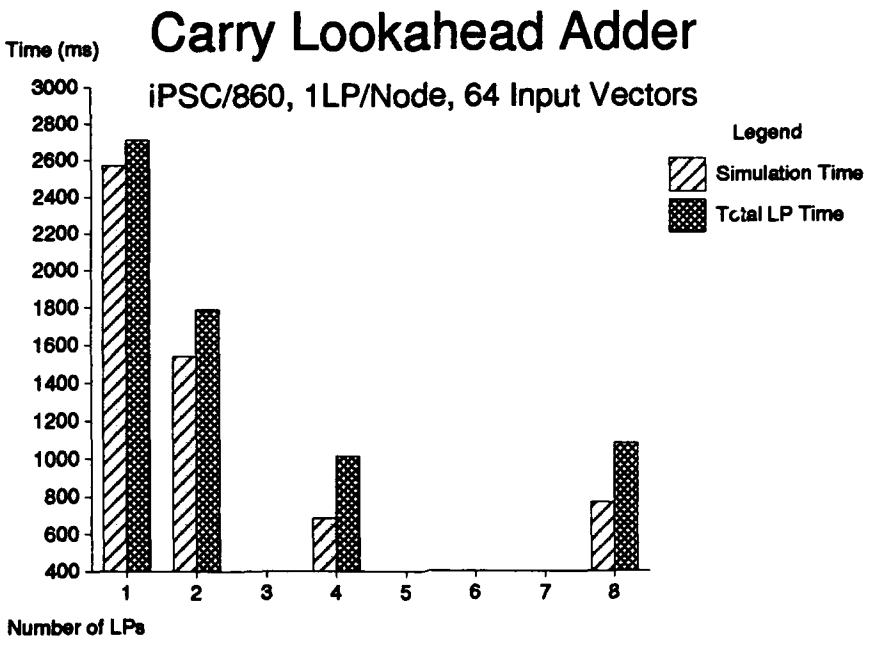


Figure 48. Performance of the Carry Lookahead Adder with 64 Input Vectors Applied (iPSC/2)



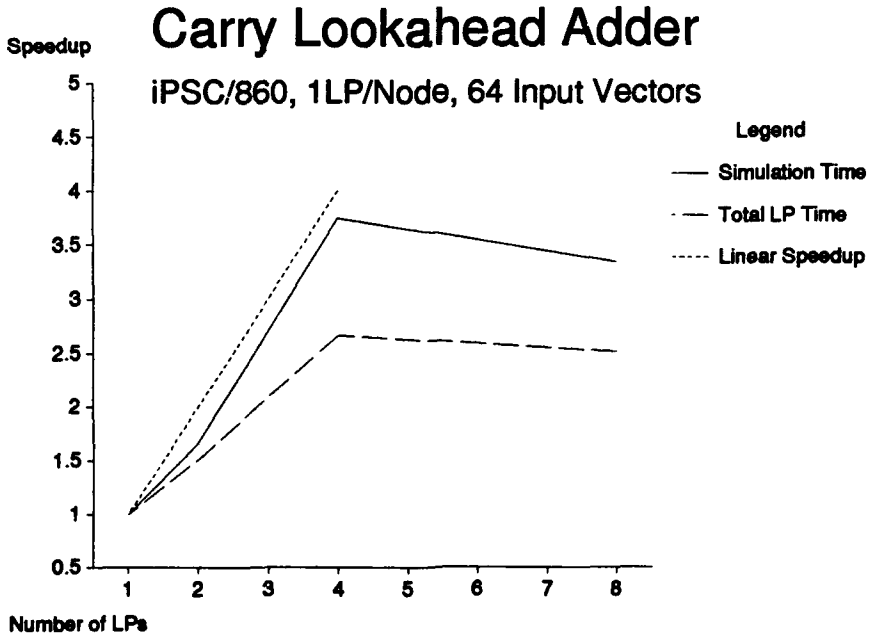


Figure 49. Performance of the Carry Lookahead Adder with 64 Input Vectors Applied (iPSC/860)

When VLSI designers test their circuits with a large number of input vectors, they are likely to automate this procedure by creating test pattern generators in VHDL. This is *not* what is simulated here. The setup for VSIM "hardwires" the input signals, through VHDL source code, at the beginning of the simulation. In this way, the active list is loaded with all input signal changes at the beginning of the simulation. With automatic test pattern generation, a behavior is created to periodically generate an input signal change, according to the rules specified in the VHDL source code. Therefore, these signal changes posted to the active list are done throughout the simulation, and not all at the beginning. Automatic test pattern generation is not implemented in this research effort.

5.7 Multitasking LPs on one Physical Processor.

The Intel 80836 processors of the iPSC/2 allow multiple processes. The carry lookahead adder and wallace tree multiplier were simulated on one node with one, two, four, and eight LPs. Results are shown in Figures 50 and 51, respectively. Note that speedups of slightly more than one are achieved with two- and four-LP simulations. These speedups are even greater as the number of input vectors are increased.

It has already been shown that one benefit of partitioning circuits is reduced active list search and post time. Clearly in sequential simulations, performance could be improved if the active list search and post time were reduced. This is inherently a part of the parallel simulation paradigm for VSIM. Improving the sequential algorithm makes all parallel configurations run faster—and it increases the challenge of achieving relative speedup through parallelization, as is the case with using faster processors like those used in the iPSC/860.

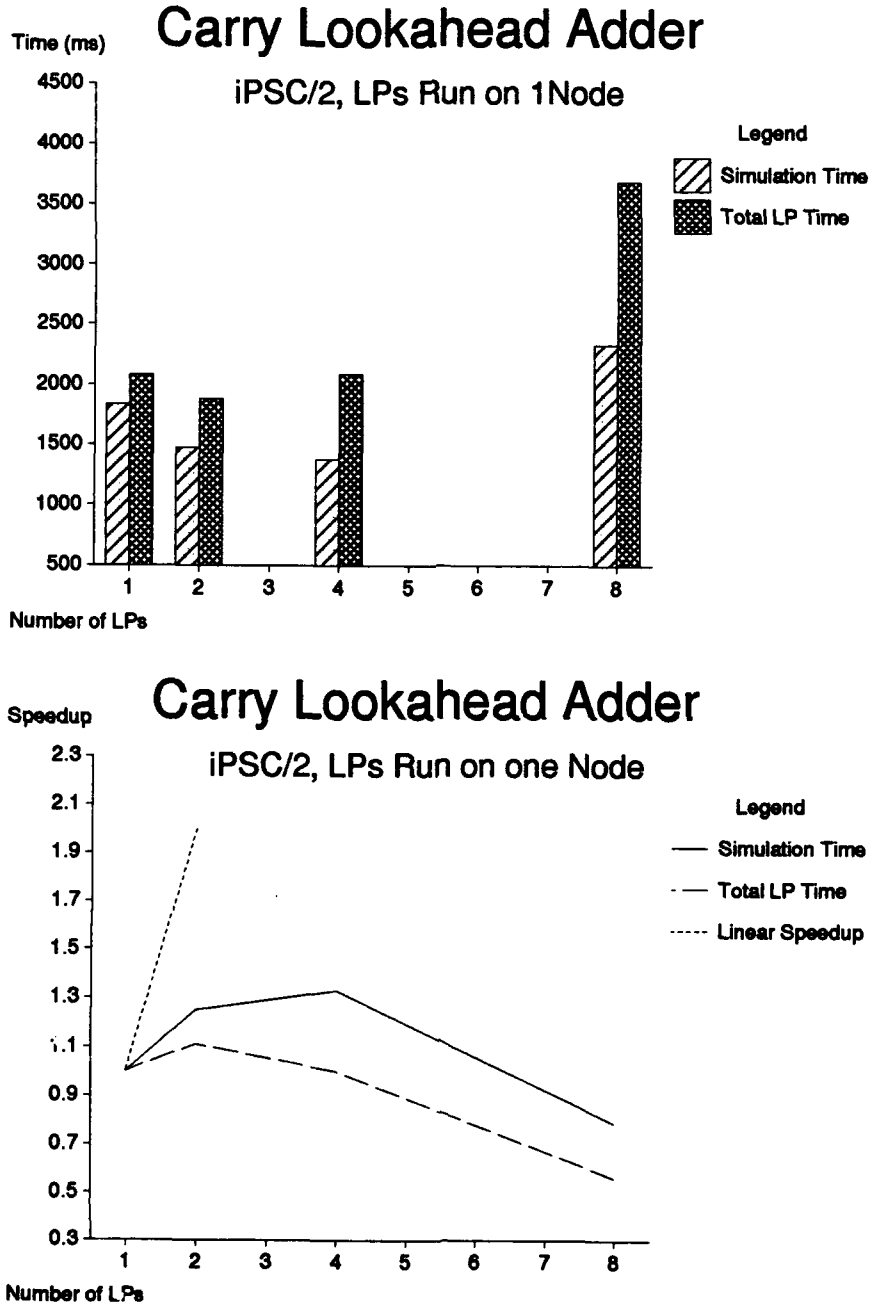


Figure 50. Performance of the Carry Lookahead Adder with all LPs Run on One Node (iPSC/2)

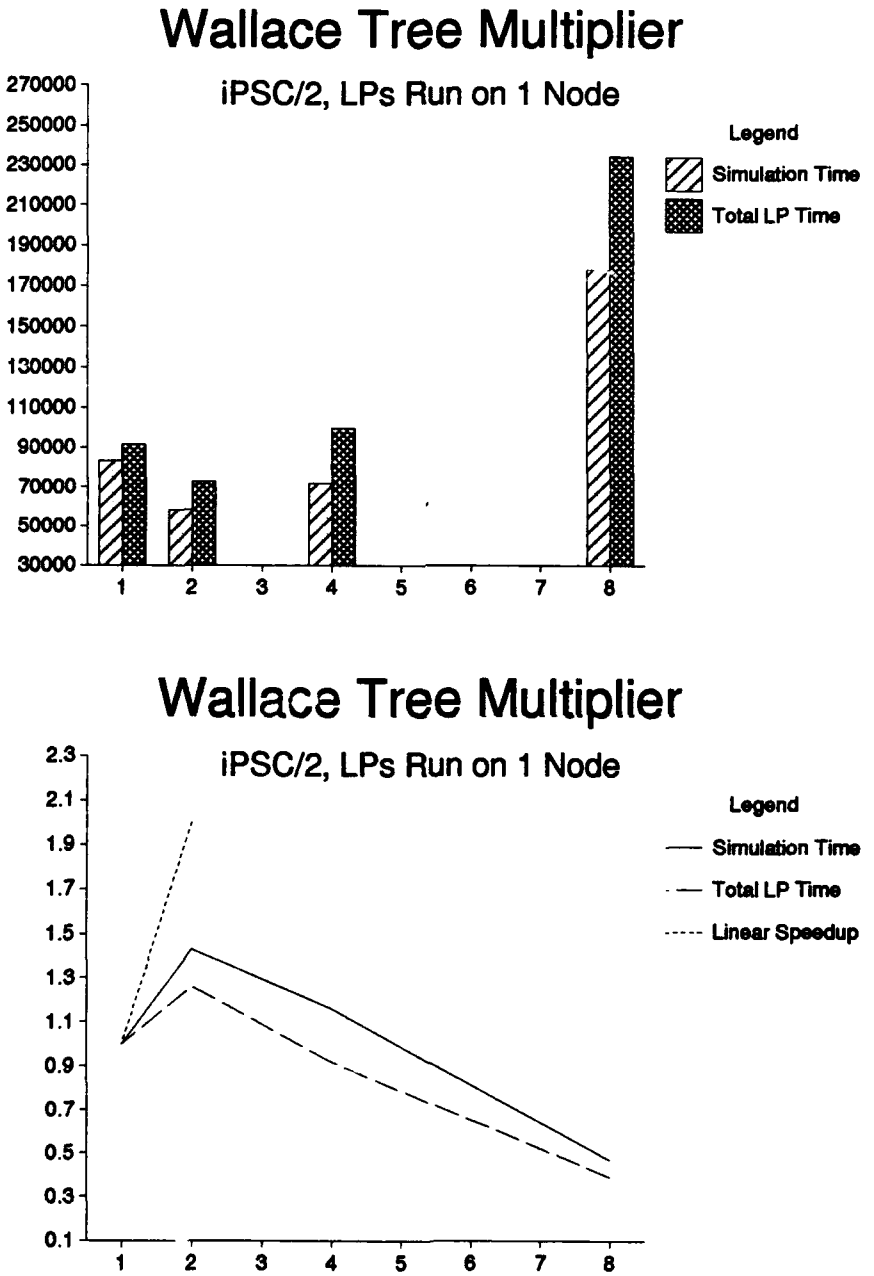


Figure 51. Performance of the Wallace Tree Multiplier with all LPs Run on One Node (iPSC/2)

5.8 Performance with Output Enabled.

All reported performance data is with output turned off, i.e., signal changes are not reported. Unfortunately, VSIM either reports all signal changes or no signal changes. Commercial simulators, like Intermetrics VHDL, allow the user to specify which signals to report.

With output enabled, each LP writes every signal change to an `lp.out` file. Since the hypercube nodes share the file system with each other and the host, this means much greater simulation time for operating system contention and file management. Execution time is significantly increased, and file I/O overwhelms the benefits of parallelization.

VI. Conclusions/Recommendations.

6.1 Research Summary.

Many circuit designs are too complex to be simulated with VHDL in a reasonable amount of time. In an effort to improve VHDL's performance, an environment is created to simulate hierarchical structural VHDL circuits in parallel on Intel Hypercube architectures.

The output from Intermetrics VHDL compile and model generate phases is transformed into code compatible with AFIT's parallel simulator. The simulator can run sequentially or in parallel on the Intel iPSC/2 and iPSC/i860. Logic gates and system behaviors are partitioned among the processors, and signal changes are shared via event messages.

The transformation and parallel simulation tools are demonstrated using three small adders: an 8-bit carry save, an 8-bit carry propagate, and an 8-bit carry lookahead. Two larger circuits are also demonstrated: a 16-bit bit/byte shifter and an 8x8 wallace tree multiplier.

No attempt is made to find *optimal* partitioning strategies; however, speedups are obtained for some configurations.

6.2 Conclusions.

With the parallel VHDL simulator, much research can now be accomplished with respect to partitioning algorithms, computation/communication balancing, etc. However, the following general observations can be made about parallel simulations of structural VHDL simulations:

- *Large circuits have a better chance to exhibit speedup.* Large circuits mean more behaviors. More behaviors mean larger active lists, which contributes to increased computation on each LP. However, a poor partition can inhibit speedup as larger active lists also correspond to increased communications. If feedback is imposed among LPs, a great number of null messages are generated to avoid deadlock. Increasing communications reduces speedup.

- *Balancing computation and communication times is hardware dependent.* The node processors of the iPSC/2 are Intel 80386 processors, while the iPSC/860 uses much faster i860 processors—which corresponds to less computation time. Therefore, a good circuit partition on the iPSC/2 may not be as effective on the iPSC/860.
- *SPECTRUM overhead is not a factor in large circuit simulations.* It was noted, however, that for small simulations, the overhead initializing SPECTRUM reduced the performance of the overall simulation. For larger circuit simulations, SPECTRUM overhead is essentially constant regardless of circuit size or configuration.
- *Performance of all simulations can be improved if active list management were improved.* One reason for obtaining speedup was reduced active list search and post times due to partitioning the behaviors, and implicitly, their output signals.

The most important conclusion is *large structural VHDL circuits can be simulated and run with speedup on the Intel hypercubes.*

6.3 Recommendations for Further Research.

6.3.1 Parallel Simulation Recommendations. The interesting work to be done in the future involves experimenting with the parallel simulation protocols and partitioning algorithms. Some suggested areas of interest are

- *Try various simulation protocols.* Since SPECTRUM is now the underlying testbed, a number of existing filters can be examined for their compatibility with VSIM.
- *Create a Time Warp version of VSIM.* Time Warp requires state-saving. The state of VSIM is identified by the simulation clock, the active list, and the global address space for signal values. If the address space were more efficiently "packed," then saving state would require much less overhead. Currently, each signal value ('0' or '1') is inefficiently stored in a 32-bit

word in memory. Packing these values aids in memory reduction, but may inhibit future enhancements to the VHDL subset—such as implementing signals as integers instead of bits, etc.

- *Determine effective partitioning strategies.* This is the subject of much research in industry and academia. To this extent, AFIT has begun work on a VHDL graph tool that reads the VSIM behavior numbers and relationships, and generates (among other things) behavior-to-LP mapping files.
- *Run simulations on larger parallel processors.* With the automation of intermediate C code translation and circuit partitioning, much larger circuits can be simulated. Simulating on larger parallel processors will aid in providing more concurrency and greater speedup.

6.3.2 Improving the Postprocessor. Currently, the postprocessor expects there to be *one* configuration description for each simulation. If configurations are broken into multiple, hierarchical descriptions, then the corresponding intermediate C code is ***significantly*** smaller. On page 95 of Appendix B, two ways to use the postprocessor on large VHDL circuits are discussed:

- Run `plex` directly on each C code description generated in the model generate phase.
- Reconstruct the VHDL circuit using hierarchical configuration descriptions.

If hierarchical configuration descriptions are used, then the user must identify the include files by examining the intermediate code before it is filtered. Automation of this function should be included as an expansion to the postprocessor.

6.3.3 Expanding the VHDL subset. The two most beneficial enhancements to the subset of circuits that VSIM can simulate are resolution functions and wait statements.

With support of resolution functions, a vast number of existing structural VHDL circuit designs can be acquired and tested. A suggested method for adding this to the subset is

1. Create a small circuit that uses a resolution function.
2. Extract the corresponding intermediate C code representation of the function.
3. Identify external data structures and function calls used in the code.
4. Determine if VSIM can support the intermediate representation.
5. If VSIM does not support the intermediate representation, design the necessary support routines and/or data structures, using Intermetrics' simulator source code as a guide.

This process can also be used to implement automatic test pattern generation and multi-valued logic.

The first step to simulating behavioral VHDL circuits is implementation of wait statements. A suggested method for adding wait statements is

1. Create simple processes with `wait`, `wait for`, `wait on`, and `wait until` statements.
2. Extract the corresponding intermediate C code and identify the methods and data structures as suggested for resolution functions.
3. Using Intermetrics' as a guide, build queues for waiting processes. If processes are allowed to "wait on" events, then execution of events that satisfy the wait condition can schedule the waiting processes (behaviors).

6.3.4 Other Recommendations.

6.3.4.1 Considerations for Generating Output. Change VSIM to report only the signal changes specified by the user. When an Intermetrics report is generated, only the signals of interest are reported, based on the user's specification in a "report control language" file. When VSIM executes with output enabled, every signal change is recorded in each LP's output file (if the LP "owns" the behavior that caused the signal change). Since the nodes of the Intel Hypercubes share the same file system, this causes a significant decrease in performance when output is enabled.

It would be beneficial to specify only the signals of interest for two reasons. First, it is how commercial simulators handle output, as in Intermetrics' case. Second, parallel performance with output enabled will improve with the reduced file contention.

This can be accomplish a number of ways. For instance, Intermetrics' report control language procedures could be studied and emulated in VSIM. A simpler approach would be to modify VSIM to compare each signal name with a list of signals of interest. The list could be built from a user file at the start of the simulation. If the changing signal is in the user-specified list, then the change is recorded in the output file and its new signal value is updated in memory. Otherwise, the change is *not* recorded in the output file; however, the new signal value is still updated in memory.

6.3.4.2 Design Method for VHDL Circuits. Design circuits hierarchically, using hierarchical configuration files. Hierarchical configurations are better for two reasons. First, as already discussed, the corresponding intermediate C code is more likely to compile on the hypercubes without running out of memory. Second, hierarchical circuit descriptions (vs. large, flat descriptions) provide insight into possible circuit partitionings by identifying groups of functionally related components. For example, a multiplier that uses sets of adders could be partitioned by assigning the components that make up each adder to the same LP.

Appendix A. *Definitions*

A.1 Discrete-Event Digital Simulation Definitions.

The following terms are used to discuss discrete-event digital simulation:

Component Any subsystem of a circuit that can be modeled as an entity, regardless of the level of hierarchy. For example, an AND gate, an arithmetic/logic unit, etc.

Entity Any component in the system which requires representation in the model (2).

Event Any action that causes the simulation model to change from one state to another (15). Typical events include the changing of any process's state variables, the arrival of a message at a process, or the transmission of a message from one process to another.

Message State information transmitted among processes.

Model An abstract representation of a physical system (2). There may be a number of models for a given system. For example, a digital circuit can be modeled by a gate-level schematic diagram, a block diagram, a dataflow graph, etc.

Process The succession of states of an entity over time (26:136). A logical process (LP) is the model's representation of a physical process (PP) in the system (7:198-199). It is common to refer to an entity as a process, although, strictly speaking, there is a distinction in the meanings.

State A collection of variables that describes the condition of an entity or system at any given time (26:136).

System The real-world process to be modeled and simulated, e.g., an electronic circuit (2).

A.2 VHDL Definitions.

The following VHDL terms are used in this thesis:

Architectural Body The description of the internal behavior or structure of a design entity (10:2-11). A structural description defines an architecture by what its subcomponents are and how the subcomponents are connected to each other (22:107). A behavioral description is used at the lowest level of decomposition and shows how the entity transforms inputs to outputs (10:2-11). See Figure 52 for an example of the relationships among behavioral and structural circuit descriptions.

Block A block may be used to define a subsystem of an architecture description (20). Blocks may be nested, and they may run concurrently.

Component The building block of hardware description, at any level of hierarchy. For example, an AND gate, a register, a chip, or a circuit board (22:18).

Design Entity The discrete system used to model a digital device. It defines the inputs and outputs of a hardware design and performs a well-defined function (22:10). A design entity may represent an entire system, a sub-system, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between (10:2-11). A design entity consists of an entity declaration and an architectural body (22:10).

Design Hierarchy The result of successive decomposition of a design entity into components. It also binds those components to other design entities that may be decomposed in like manner. Taken together they represent a complete design. Such a collection of design entities is called a design hierarchy (10:2-12).

Entity Declaration The entity declaration defines the component's interface to the external environment; it specifies the ports of the entity in which data may flow in and out (22:18).

External Block The top-most block in a hierarchy. This block is the design entity itself, and it defines the interface of the design entity to the external environment (10:2-12).

Inertial Delay Delay-type representing components which require the value on inputs to persist for a given time before the component responds(22:71).

Model The elaboration of the design hierarchy in the VHDL simulation environment. The model is executed to simulate the behavioral or structural design of the circuit under test (10:2-12).

Port A signal that appears in the interface list of an entity declaration (10:2-12). Also, a port is a component's external interface, the point where data flows into and out of the component (22:18).

Process A collection of operations applied to signals. The operations are sequential descriptions of component behavior. Processes are said to run *concurrently*. Therefore, VHDL descriptions can be thought of as a set of independent programs running in parallel (22:9).

Signal An object that holds a value and directly corresponds to some type of metal interconnection within a circuit (10:2-12). Signals define the pathways among processes (22:9).

Transport Delay Delay-type representing an output which always occurs regardless of the time duration of the input signals (22:71).

Note that some terms, like *entity*, *model*, and *process* have different meanings, depending on the context—classical simulation or VHDL. The reader is cautioned to interpret each term with respect to its context.

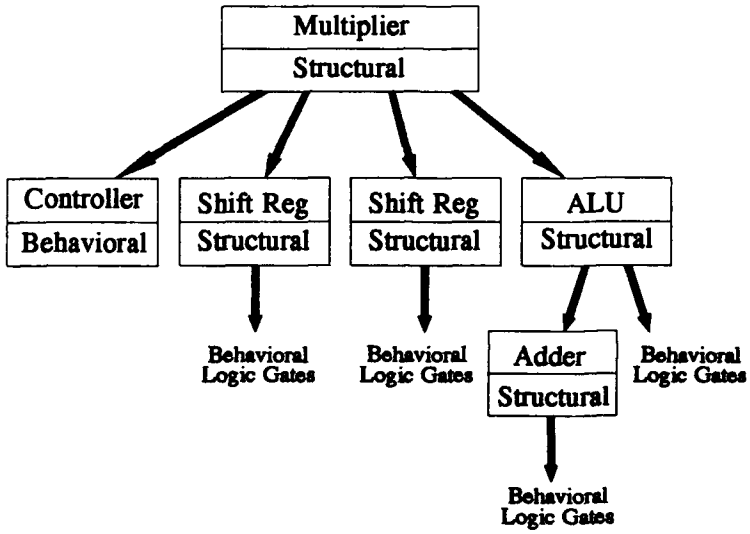


Figure 52. Example of the Relationships Among Behavioral and Structural Circuit Descriptions in a Mixed-Level Design

Appendix B. *AFIT Parallel VHDL User's Guide*

B.1 Overview.

B.1.1 Introduction. When a VHDL circuit is compiled with the Intermetrics VHDL toolset, the intermediate C code must be intercepted and transformed to be linked with AFIT's parallel VHDL simulator (VSIM). VSIM runs sequentially on a single processor, or in parallel on the Intel iPSC/2 and iPSC/i860 Hypercubes. For parallel simulations, VSIM runs over SPECTRUM—a testbed that provides an interface between user applications and the parallel processing environment. The subset of VHDL circuits that can be simulated with VSIM includes structural descriptions of logic gates and simple processes.

B.1.2 Process. The process for developing and running parallel VHDL circuit simulations is as shown in Figure 53. In general, the following steps must be taken:

1. Write VHDL source code to describe the circuit to be simulated.
2. Compile, Model Generate, and Build using Intermetrics' VHDL tools.
3. Use the postprocessor, **pbuild**, to generate C code that can run with VSIM.
4. Compile and run the C code with VSIM on a sequential processor.
5. Use **vmap** to generate behavior id numbers and dependencies.
6. Decide on partitioning strategy and create logical process (LP) dependency file, **lpx.arcs**, and behavior-to-LP mapping file, **lpx.map**.
7. Compile with VSIM and SPECTRUM on the Hypercube and run the simulation in parallel.

B.1.3 Related Files.

B.1.3.1 The Postprocessor. The postprocessor, called **pbuild**, is used to translate Intermetrics' C code into code compatible with VSIM. The files necessary for operation and maintenance of **pbuild** are shown in Table 2.

B.1.3.2 VSIM. The AFIT parallel VHDL simulator, VSIM, is comprised of two groups of files. The first group, listed in Table 3, contains all of the VSIM-specific files required for sequential operation. When the simulation is run in the sequential mode, the executable filename is generally the name of the circuit. When the simulation is run on a parallel machine, the files of Table 4 are also included, and the executable file called by the user is generally called "**host**," which loads each node of the hypercube with the appropriate node programs.

B.1.3.3 VMAP. VMAP is used to determine the behavior id numbers and dependencies. In order to use VMAP, run the simulation in sequential mode with **MAPPING** defined in **vsim.h**. Then run the output through the program called **vmap**. The files required for VMAP operation and maintenance are shown in Table 5.

B.1.3.4 Other Files. Other files related to VSIM simulations are listed in Table 6. These include the source code and headers for Intermetrics' intermediate C code, LP dependency and mapping files, output files, and some helpful scripts.

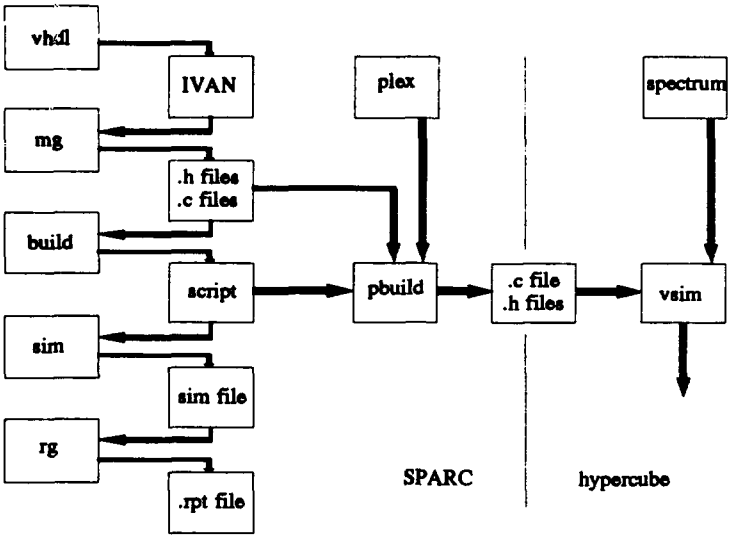


Figure 53. Overview of Parallel Simulation Session

Table 2. Files Necessary for Maintenance and Operation of the Postprocessor

File	Description
pbuild	Executable called by user, finds intermediate C code and calls plex.
plex	Executable called by pbuild, uses lexical analyzer and regular expressions to find and transform intermediate C code.
pbuild.c	Source code for pbuild.
plex.l	Lex description and rules for pattern matching.
plex.h	Header file for plex.l and plex_routines.c.
plex_routines.c	Routines called by plex.l to transform data.
stack.c	Character stack used by plex_routines.c.
stack.h	Header file for stack.c.
Makefile	Describes sequence of commands necessary for generating executables. The command "make" generates pbuild. Use "make plex" to generate plex.

Table 3. Files Necessary for Maintenance and Operation of VSIM

File	Description
vsim.h	Header file for vinit.c, vsim.c, vtools.c, and vspec.c. Modeled after Intermetrics' simutl.h.
vinit.c	Initialization routines for VSIM.
vsim.c	The main simulation loop and functions.
vtools.c	Tools provided for printing VSIM state variables and queues. Compilation is optional—only required for maintenance purposes.

Table 4. Files Necessary for Maintenance and Operation of Parallel VHDL Simulations using SPECTRUM

File	Description
vspec.c	Contains the functions that provide VSIM's interface to SPECTRUM.
vfilt.c	Contains the null-message protocol filters. Modeled after AFIT's chanclocks.c.
u_null_filt.c	Table of function-pointers to filters in vfilt.c.
globals.h	The standard header file for SPECTRUM. Modified to redefine event structure.
application.h	Included by globals.h, this file contains application-specific global information for SPECTRUM and vspec.c. Most importantly, this file is where the number of LPs are specified for a particular simulation.
lp_man.c	Provides SPECTRUM's LP-level functions.
cube2.c	Provides hardware interface for lp_man.c.
cube2.h	Header file for cube2.c and host2.c.
host2.c	Host program used to load nodes and start simulation.

Table 5. Files Necessary for Maintenance and Operation of VMAP

File	Description
vmap	Executable used to generate mapping.
vmap.c	Source code for vmap.
list.c	Linked-list functions for vmap.c.
list.h	Header file for list.c.
makefile	Describes command sequence necessary for generation of vmap.

Table 6. Other Files

File	Description
plex.log	Report generated by postprocessor.
(ckt).c big_(ckt).c	Postprocessor output file, named by the user when invoking Big C file containing intermediate C code prior to transforming with plex.
	pbuild. This is the intermediate C code.
FN*	Header files included by (ckt).c.
lpx.out	Output files for parallel simulations. For example, "lp2.out" corresponds to the output of LP2. In sequential simulations, the output is sent to "stdout."
lpx.arcs	LP dependencies and output delays, generated by the user.
lpx.map	Behavior-to-LP mapping description, generated by the user.
logx	SPECTRUM reports from each LP.
sgrep	Script used to extract signal changes from VSIM's output and sort by time and signal name.

```
# The following setup Intermetrics' VHDL
set path = ($path /usr/vhdl/bin)
setenv VHDL_BIN /usr/vhdl/bin
setenv VHDL_LIBROOT /usr/vhdl/shiplib
setenv VHDL_COMMON /usr/vhdl/common
setenv VLS_HELP_FILE /usr/vhdl/common/help.txt
```

Figure 54. Section of .cshrc File for Setting up Intermetrics VHDL in the AFIT VLSI Lab

---

B.2 Implementation.

B.2.1 Introduction. This section describes how to create and run parallel VHDL simulations with VSIM. The following section illustrates these steps with an example.

B.2.2 Generating VHDL Source Code. The first step is to create the VHDL circuit description in one or more **.vhd** files. VSIM can simulate structural descriptions of logic gates and other simple processes. Circuits are created the same way as for Intermetrics' circuits, with the following limitations:

B.2.2.1 VHDL Source Code Limitations for VSIM. Signals can be bits or bit-vectors, but bit-vector inputs must be described one bit at a time, e.g., `Bus(0) <= '1' after 10 ns;`.

Processes should be one-line descriptions (`Out1 <= In1 AND In2 after gate_delay;`); however, multiline processes—delimited by **begin** and **end process** may be used provided they either wait on all signals, or the process terminates after first use, i.e., it contains a **wait;** statement at the end of the process block.

It is uncertain how functions and procedures will act in VSIM. For example, functions to describe multi-valued logic—or signal resolution—have not been tested. Their implementation may or may not be trivial; however, the file **vsim.h** would most likely have to be modified to include the proper macros and type-definitions. Intermetrics' file, **simutl.h**, was used as a baseline for **vsim.h**, with much of the (at the time) unnecessary data removed.

B.2.3 Setting up a User Library for Circuit Models. In order to use Intermetrics VHDL simulator, the following environment variables must be defined in the user's **.cshrc** file: **VHDL_BIN**, **VHDL_LIBROOT**, **VHDL_COMMON**, and **VLS_HELP_FILE**. Intermetrics' VHDL is available on in the VLSI lab, and is in the process of being installed on **aphrodite** in the Parallel Simulation Lab.[1] The correct environment setup for using Intermetrics VHDL in the VLSI lab is shown in Figure 54.

Once the correct environment variables are set, the user creates a work library by using **vls**, **define**, and **makelib**, as shown in Figure 55.[2] The commands **setlib** and **dir** can be used to view the current library and its contents, respectively. For the most convenience when using the postprocessor, the user should give the work directory the same name as his or her userid (**${LOGNAME}**).

[1] Also, **hercules** on the VAX cluster has a version of Intermetrics' VHDL simulator.

[2] Should the error "**VHMMARKRELEASEERROR**" ever be raised by Intermetrics, the only solution is to delete the complete user directory using **delete-user**.

```
lovelace% vls
Standard VHDL 1076 Support Environment Version 2.1b - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

VLS>makelib -dir=/usr/vhdl/shiplib/tbreeden <<tbreeden>>
VHDVLS-I-CREATED_LIB - Library <<TBREEDEN>> successfully created.
VLS>define work <<tbreeden>>
VLS>setlib <<tbreeden>>
VHDVLS-I-DEFAULT_LIBRARY - Default library is <<TBREEDEN>>.
VLS>dir
VHDVLS-I-NO_UNITS - No units found in <<TBREEDEN>>.
VLS>exit
lovelace%
```

Figure 55. Example Initialization of Intermetrics VHDL

B.2.4 Compiling, Model Generating, and Building. Every **.vhd** file is compiled individually by using the command **vhdl**[3], such as

```
vhdl nand_gate.vhd
```

To "model generate" the specific entity/architecture pairs, the command **mg** is used; however, the debug switch **-debug=cknd** is added as so:

```
mg '-debug=cknd nand_gate(simple)'
```

This generates the required **.c** and **.h** files for the postprocessor. This debug switch is also used in the "build" phase, using the command **build**:

```
build '-debug=cknd -replace -ker=etdff etdff_config'
```

In this manner, the compilation script is generated; then, the postprocessor can determine the correct files and their order required for compilation.

B.2.5 Extracting and Transforming Intermediate C Code. In order to transform the intermediate C code generated during the "model generate" phase above, type

```
pbuild scriptname outputname.c
```

where **scriptname** is the compilation script generated during the "build" phase, and **outputname.c** is the user's name for the transformed C file.

Finally, send the new **.c** file, and all the header files it includes, to the target machine for sequential (and/or parallel) simulation. The header files are named in the top of the new **.c** file, and can be found in the user's work directory.[4]

[3] The **.vhd** extension is optional.

[4] Unless the user has compiled them into another directory through commands in the VHDL source code.

B.2.5.1 Handling Difficult Files. When large circuit simulations are compiled under Intermetrics, the corresponding C code generated by the postprocessor may be too large to compile on the Intel Hypercubes. There are two methods of getting around this:

- Run **plex** directly on each C code description generated in the model generate phase.
- Reconstruct the VHDL circuit using hierarchical configuration descriptions.

If **plex** is run directly on each C code file, the resulting output can be compiled into separate object files and linked together on the Hypercubes. Currently, the 16-bit shifter on the Intel i860 is constructed in this manner. The best way to do this is to first try using **pbuild** directly. If this big C file does not compile, then run **plex** on each intermediate C file. The "main" file—found by examining the compilation script—can either be edited by hand, or can be pulled in from the big C file generated by running **pbuild**.

When VHDL structural circuit descriptions are build hierarchically, using hierarchical configuration descriptions, the size of intermediate C code resulting from model generating the overall configuration file is significantly reduced. For example, a Wallace Tree multiplier was designed in this manner. Even though the multiplier has about 20 times more logic gates than some other VSIM/VHDL circuit descriptions, the amount of C code is about the same. The postprocessor does not, however, catch all of the include directives necessary for compilation. These can be found and inserted "by hand" by inspecting each intermediate C code representation of each configuration.

For example, here are portions of the intermediate C code for the wallace tree multiplier prior to transformation:

```
/* CGF_WALLACE_TB */

#include "simutl.h"
#include "fn26"

static char Z000006B_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 71, 70, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 66, 0 };

#include "/usr/vhdl/shiplib/tbreeden/FN21712"
#include "/usr/vhdl/shiplib/tbreeden/FN21682"
.
.
.
/* CFG_WALLACE_TREE_2 */

#include "simutl.h"
#include "fn26"

static char Z0000068_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 70, 71, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 82, 69, 69, 95, 50, 0 };

#include "/usr/vhdl/shiplib/tbreeden/FN21682"
#include "/usr/vhdl/shiplib/tbreeden/FN21667"
#include "/usr/vhdl/shiplib/tbreeden/FN21607"
#include "/usr/vhdl/shiplib/tbreeden/FN2665"
```

```
#include "/usr/vhdl/shiplib/tbreeden/FN21597"
.
.
.
/* CFG_WALLACE_TREE_1 */

#include "simutl.h"
#include "fn26"

static char Z0000065_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 70, 71, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 82, 69, 69, 95, 49, 0 };

#include "/usr/vhdl/shiplib/tbreeden/FN21667"
#include "/usr/vhdl/shiplib/tbreeden/FN21652"
#include "/usr/vhdl/shiplib/tbreeden/FN21622"
#include "/usr/vhdl/shiplib/tbreeden/FN2665"
#include "/usr/vhdl/shiplib/tbreeden/FN21607"
#include "/usr/vhdl/shiplib/tbreeden/FN21597"
.
.
.
```

After transformation, only the two include directives from the top of the first file are included in the transformed file:

```
/* CGF_WALLACE_TB */

#include "vsim.h"

static char Z000006B_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 71, 70, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 66, 0 };

#include "FN21712"
#include "FN21682"
.
.
.
/* CFG_WALLACE_TREE_2 */

static char Z0000068_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 70, 71, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 82, 69, 69, 95, 50, 0 };
.
.
.
/* CFG_WALLACE_TREE_1 */
```

```
static char Z0000065_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 70, 71, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 82, 69, 69, 95, 49, 0 };
.
.
.
```

By examining the initial intermediate C code, the user can then put all of the include directives in the top of the transformed file, as shown:

```
/* CGF_WALLACE_TB */

#include "vsim.h"

static char Z000006B_trcbck []= {
 60, 60, 84, 66, 82, 69, 69, 68, 69, 78, 62, 62, 67, 71, 70, 95, 87, 65, 76,
 76, 65, 67, 69, 95, 84, 66, 0 };

/* Added by TAB, 2 Oct 92 */

#include "FN21712"
#include "FN21682"
#include "FN21667"
#include "FN21607"
#include "FN2665"
#include "FN21597"
#include "FN21652"
#include "FN21622"
#include "FN21637"
#include "FN2635"
#include "FN2645"
```

B.2.6 Running VSIM on a Sequential Machine. As is the case with Intermetrics' simulator, each gate is dynamically assigned a behavior number in VSIM. VSIM must first be run in sequential mode in order to see how the behaviors are numbered. To do this, define **MAPPING** in **vsim.h**. This way, when the simulation is run, VSIM reports which behaviors are executing and which behaviors are consequently scheduled because of that execution, i.e., *dependent* behaviors.

To specify that the simulation is to be sequential, define **SPARC** in **vsim.h** or in the makefile.[5] Also, if signal change output is desired, define **OUTPUT** in **vsim.c**.

Now compile **vinit.c**, **vsim.c**—and optionally **vtools.c**—with the intermediate C code circuit description, and run the simulation.

[5] Although the name is SPARC, sequential simulations may be compiled and run on the hypercube host or most likely any other machine with a C compiler, if desired.

```
0                   # LP index
2                   # Number of input LPs
1 2                 # LP indices of input LPs
0 0                 # Polling frequencies of input LPs
0 0                 # Offset of polling frequency
2                   # Number of input lines
1 2                 # LP number for each input line
2                   # Number of output LPs
2 3                 # LP indices of output LPs
2                   # Number of output lines
2 3                 # LP index for each output line
3000000 5000000     # Minimum delays for each output line
```

Figure 56. Example Format for One LP in an lpx.arcs File

B.2.7 Generating Partitioning Strategies. After running the sequential simulation with mapping turned on, the output can be run through **vmap** to generate a list of behaviors and dependencies. This step is not necessarily required; **vmap** was created to generate an output file that can be used in future research related to circuit partitioning strategies. If the simulator output is in the file **etdff.raw**, then type

```
vmap etdff.raw etdff.map
```

to generate the mapping file, **etdff.map**. If a list of signal changes is desired, the script **sgrep** is provided to pull out and sort the signal changes by time and signal name as so:

```
sgrep etdff.raw etdff.out
```

If desired, this data can be compared with the output of Intermetrics' simulator in order to check for correctness.

The user must now decide how to partition the circuit among LPs.[6] Once the partition is determined, an **lpx.arcs** file must be created to define the LP dependencies and output delays. SPECTRUM uses this file. Also, VSIM reads an **lpx.map** file created to map each behavior to an LP. *These two files must be created with great care.* VSIM and SPECTRUM assume the user knows what he/she is doing, and in most cases, they faithfully try to comply. The **lpx.arcs** and **lpx.map** formats are shown in Figures 56 and Table 7, respectively.[7]

[6]The scheme for distributing LPs among processors is defined at run time.

[7]The polling frequencies and offsets in the .arcs files are not used with the current filters, so zeroes can be entered. If the number of input or output LPs or lines is zero, the other entries relating to those LPs or lines are omitted. The comments shown in Figure 56 and Table 7 are not included. Although more than one input line is permitted from each LP, communications from one LP to another in VSIM can be considered to occur on one input line. Delays are in femptoseconds.

Behavior	LP Number
0	0
1	0
2	1
3	2
⋮	⋮

Table 7. Example Format for the lpx.map File

B.2.8 Running VSIM on a Parallel Machine. Before compiling, be sure the desired number of LPs and the LP input file is specified in `application.h`. If `application.h` is in the user's `~/spectrum` directory, this can be done by typing `setlps x`, where `x` is the number of LPs desired.[8]

Remove the `MAPPING` and `SPARC` definitions and compile the intermediate C code with `vinit.c`, `vsim.c`, `vtools.c` (optional), `lp_man.c`, `cube2.c`, `u_null_filt.c`, and `vfilt.c`. This generates the executable program that is loaded on the processors and represent each LP. Note that on the iPSC/2, more than one LP may be loaded on a processor due to the multitasking capabilities of the Intel 80386 processors. On the iPSC/i860, however, the number of LPs must match the number of processors.

The `host` program is used to load the LPs on the processors. It's created by compiling `host2.c`.

When the necessary files are compiled, type `host`. Among other things, it asks for the name of the program to load, the number of processors desired, and the number of LPs. *The number of LPs must match the number specified in* `application.h`. If not, the program "bails out."

Each LP reports when it is finished running, and after every LP has completed, the `host` program reports time and message statistics. If `OUTPUT` was defined in `vsim.c` and `vspec.c`, the output can be found in the group of files labeled `lpx.out`, where `x` is the LP number. Timing information can be found in `logx`, `x` again being the specific LP number. If `DEBUG` or `REPORT` is set to '1' in `globals.h`, the `logx` files report more information than humanly consumable. This comes from `lp_man.c` and `cube2.c`. Usually, filters also have `DEBUG` output, but this author chose to leave it out of `vfilt.c` for simplicity.

Finally, the `lpx.out` files can be concatenated (provided `OUTPUT` was defined) and `sgrep` can be invoked to generate a file that can be compared with the sequential output.

B.3 Example: An Edge-Triggered D Flip-Flop.

B.3.1 Introduction. This section goes through an example using the edge-triggered D flip-flop of Figure 57 on page 114. The VHDL source code is compiled and run on a SPARC station in the AFIT VLSI lab, sequential VSIM is run on a SPARC station in the Parallel lab, and output is compared to Intermetrics' output. Finally, parallel simulations are executed on the Intel iPSC/2 Hypercube—one simulation with two LPs and no feedback, the other with three LPs and feedback between two of the LPs.

All figures referenced in this example are located at the end of the document.

[8] The program `setlps` simply changes any integer in the first eight lines of `application.h` to the specified integer. For more than nine LPs, the user must modify the file directly.

B.3.2 VHDL Source Code. First, two- and three-input NAND gates are created, as shown in Figure 58 on page 115. For this example, these entity/architecture descriptions are located in a user file called "**nand_nor.vhd**."

Next, the NAND gates are structurally connected to form the edge-triggered D flip-flop. This description, shown in Figure 59 (page 116), is in a file called "**et_dff.vhd**."

To test this circuit, a "test bench" is written to apply input signals and receive output signals. This file, **et_dff_test_bench.vhd**, is shown in Figure 60 (page 117), and the schematic is shown in Figure 61 on page 118.

The last VHDL source file is the "configuration file," which structurally connects the components, as shown in Figure 62. This file is called **et_dff_config.vhd**.

Intermetrics uses a "report control language" to generate a report of desired signal changes. The file for this example, **et_dff.rcl** is shown in Figure 63.

B.3.3 Compiling, Model Generating, Building, and Simulating under Intermetrics. A script like that of Figure 64, on page 120, can be run to compile, model generate, build, and simulate the circuit with Intermetrics VHDL. Notice the placement of "**-debug=cknd**" in the **mg** and **build** phases. This generates the intermediate C code and build script required for the postprocessor, **pbuild**.

The following is an example session using the script of Figure 64:

```
lovelace.~/vhdl/et_dff>et_dff

vhdl ~/vhdl/aox_gates/nand_nor.vhd
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

vhdl et_dff.vhd
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

vhdl et_dff_test_bench.vhd
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

vhdl et_dff_config.vhd
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

mg '-debug=cknd nand_gate(simple)'
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

Object_file : /home/inter/shiplib/tbreeden/FM272.o
H file      : /home/inter/shiplib/tbreeden/FM273
C file      : /home/inter/shiplib/tbreeden/FM274.c
mg '-debug=cknd three_input_nand_gate(simple)'
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
```

```
Object_file : /home/inter/shiplib/tbreeden/FN282.o
H file      : /home/inter/shiplib/tbreeden/FN283
C file      : /home/inter/shiplib/tbreeden/FN284.c
mg '-debug=cknd et_dff(structural)'
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

Object_file : /home/inter/shiplib/tbreeden/FN2102.o
H file      : /home/inter/shiplib/tbreeden/FN2103
C file      : /home/inter/shiplib/tbreeden/FN2104.c
mg '-debug=cknd et_dff_test_bench(structural)'
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

Object_file : /home/inter/shiplib/tbreeden/FN2112.o
H file      : /home/inter/shiplib/tbreeden/FN2113
C file      : /home/inter/shiplib/tbreeden/FN2114.c
mg '-debug=cknd -top et_dff_config'
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

Object_file : /home/inter/shiplib/tbreeden/FN2117.o
H file      : /home/inter/shiplib/tbreeden/FN2118
C file      : /home/inter/shiplib/tbreeden/FN2119.c
build '-debug=cknd -replace -ker=et_dff et_dff_config'
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

Kernel com file is /home/inter/shiplib/tbreeden/FN2122
sim et_dff
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

SIGTRAN Signal Tracing turned on
QUIESCE Quiescent state reached with no response after 512 ns

rg et_dff et_dff.rcl
Standard VHDL 1076 Support Environment Version 2.1 - 1 September 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

lovelace.~/vhdl/et_dff>
```

Here is the output from Intermetrics' simulator—found in `et_dff.rpt`:[9]

```
  TIME  |----------------------------SIGNAL NAMES----------------------------|
        |
  (NS)  |          A          B          CKT_Q_OUT         CKT_Q_BAR_OUT
```

[9]The + values are delta delays and can be considered to have a delta time value of zero.

```
     |
   0 |        '0'        '0'          '0'              '0'
   3 |
  +1 |                               '1'              '1'
   6 |
  +1 |                               '0'              '0'
   9 |
  +1 |                               '1'              '1'
  12 |
  +1 |                               '0'              '0'
  15 |                  '1'
  +1 |                               '1'              '1'
  18 |
  +1 |                               '0'              '0'
  20 |                  '0'
  21 |
  +1 |                               '1'              '1'
  24 |
  +1 |                               '0'
  50 |                  '1'
 100 |        '1'
 150 |                  '0'
 200 |                  '1'
 206 |
  +1 |                               '1'
 209 |
  +1 |                                                '0'
 250 |        '0'
 300 |                  '0'
 350 |                  '1'
 356 |
  +1 |                                                '1'
 359 |
  +1 |                               '0'
 400 |        '1'
 450 |                  '0'
 500 |                  '1'
 506 |
  +1 |                               '1'
 509 |
  +1 |                                                '0'
```

B.3.4 Using the Postprocessor to Generate Intermediate C Code. Notice that after the model generate phase, Intermetrics reported a "Kernel com" file, **FM2122**. This is the compilation build script **pbuild** uses to build the intermediate C code: **et_dff.c**, as shown below. The report shown is always written to a file called **plex.log**.

```
lovelace.~/vhdl/et_dff>pbuild FM2122 et_dff.c
cp /home/inter/shiplib/tbreeden/FM2119.c big_et_dff.c
```

```
cat /home/inter/shiplib/tbreeden/FN284.c >> big_et_dff.c
cat /home/inter/shiplib/tbreeden/FN274.c >> big_et_dff.c
cat /home/inter/shiplib/tbreeden/FN2104.c >> big_et_dff.c
cat /home/inter/shiplib/tbreeden/FN2114.c >> big_et_dff.c
cat /home/inter/shiplib/tbreeden/FN2124.c >> big_et_dff.c
plex < big_et_dff.c > et_dff.c
Transformation in progress...

-----------------------------------------------

Approx lines:                              1964
Comments:                                     5
#include directives modified:                 5
#include directives removed:                13
{trace... changed to {...  :                 12
if(trceqp) tests removed:                    21
"trace" or "TRAREC" lines removed:          133
Z1xxxxxx() calls removed:                     4
Z5xxxxxx() functions modified:                1
Scalar "mksig" assignments modified:         10
Bit vector "mksig" assignments modified:      0
#ifdef MAPPING added:                         6

Other function calls removed:

        close_sigdict():          1
        m_int_type():             0
        m_real_type():            1
        pop():                   13
        push():                  13
        read_input():             1
        rmtrrec():                0
        rptstats():               1
        rpterr():                23
        Start_Nonarray_Comp():    0
        sched():                  0
        timer():                  1
        tpop():                  31

-----------------------------------------------


In addition to et_dff.c, copy all H files over to the iPSC/2.

lovelace.~/vhdl/et_dff>
```

B.3.5 Sequential Simulation with VSIM. The intermediate C code, **et_dff.c**, and the header files it includes, **FN2113**, **FN283**, **FN273**, and **FN2103** are now linked with VSIM and simulated on a sequential machine—**neptune** in this example. First, **MAPPING** is defined in **vsim.h** and

OUTPUT is defined in **vsim.c** and **vspec.c**.[10] The following makefile compiles and links for either sequential simulations (by typing **make vsim**) or parallel simulations (by typing **make ipsc** for the iPSC/2 or **make** for the iPSC/i860):

```
# SPARC macros
S_SIMPATH=/olympus3/eng/tbreeden/vsim
S_CKTPATH=/olympus3/eng/tbreeden/et_dff
S_SPECPATH=/olympus3/eng/tbreeden/spectrum
S_OBJS=${S_SIMPATH}/vsim.o ${S_SIMPATH}/vinit.o ${S_SIMPATH}/vtools.o \
       ${S_CKTPATH}/et_dff.o
S_CFLAGS=-c -w -g -DSPARC

# iPSC/2 macros
I_SIMPATH=/usr2/eng/tbreeden/vsim
I_CKTPATH=/usr2/eng/tbreeden/et_dff
MYSPECPATH=/usr2/eng/tbreeden/spectrum
UVAPATH=/usr/simulate/spectrum/uva
AFITPATH=/usr/simulate/spectrum/afit
AFIT_INC=/usr/simulate/spectrum/afit/include
FILTERPATH=${MYSPECPATH}
SPECHEADERS=${MYSPECPATH}/globals.h ${MYSPECPATH}/application.h
NODE_OBJS=${I_SIMPATH}/vsim.o ${I_SIMPATH}/vinit.o ${I_SIMPATH}/vtools.o \
          ${I_SIMPATH}/vspec.o ${MYSPECPATH}/lp_man.o ${MYSPECPATH}/cube2.o \
          ${MYSPECPATH}/u_null_filt.o  ${MYSPECPATH}/vfilt.o \
          ${I_CKTPATH}/et_dff.o

I_CFLAGS=-c -w

# iPSC/i860 macros
I8_SIMPATH=/usr2/tbreeden/vsim
I8_CKTPATH=/usr2/tbreeden/et_dff
MY8SPECPATH=/usr2/tbreeden/spectrum
UVA8PATH=/usr2/tbreeden/spectrum
AFIT8PATH=/usr2/tbreeden/spectrum
AFIT8_INC=/usr2/tbreeden/spectrum
FILTER8PATH=${MY8SPECPATH}
SPEC8HEADERS=${MY8SPECPATH}/globals.h ${MY8SPECPATH}/application.h
NODE8_OBJS=${I8_SIMPATH}/vsim.o ${I8_SIMPATH}/vinit.o ${I8_SIMPATH}/vtools.o \
          ${I8_SIMPATH}/vspec.o ${MY8SPECPATH}/lp_man.o ${MY8SPECPATH}/cube2.o \
          ${MY8SPECPATH}/u_null_filt.o  ${MY8SPECPATH}/vfilt.o \
          ${I8_CKTPATH}/et_dff.o
I860CC=icc


# other macros
CC=cc

#                       -------- iPSC/i860 ----------
```

[10] The makefile defines **SPARC**.

```
all: host8 node8

host8: ${MY8SPECPATH}/host2.o
$(CC) -o host ${MY8SPECPATH}/host2.o -host

node8: ${NODE8_OBJS}
$(I860CC) -o et_dff ${NODE8_OBJS} -node

${MY8SPECPATH}/host2.o: ${MY8SPECPATH}/host2.c ${AFIT8_INC}/cube2.h
cd ${MY8SPECPATH}; \
$(CC) ${I_CFLAGS} -I${AFIT8_INC} ${MY8SPECPATH}/host2.c

${I8_SIMPATH}/vsim.o: ${I8_SIMPATH}/vsim.c ${I8_SIMPATH}/vsim.h
                      ${MY8SPECPATH}/application.h
cd ${I8_SIMPATH}; \
$(I860CC) ${I_CFLAGS} -I${MY8SPECPATH} vsim.c

${I8_SIMPATH}/vinit.o: ${I8_SIMPATH}/vinit.c ${I8_SIMPATH}/vsim.h
                       ${MY8SPECPATH}/application.h
cd ${I8_SIMPATH}; \
$(I860CC) ${I_CFLAGS} -I${MY8SPECPATH} vinit.c

${I8_SIMPATH}/vtools.o: ${I8_SIMPATH}/vtools.c ${I8_SIMPATH}/vsim.h
cd ${I8_SIMPATH}; \
$(I860CC) ${I_CFLAGS} vtools.c

${I8_SIMPATH}/vspec.o: ${I8_SIMPATH}/vspec.c ${I8_SIMPATH}/vsim.h
                       ${MY8SPECPATH}/application.h
cd ${I8_SIMPATH}; \
$(I860CC) ${I_CFLAGS} -I${MY8SPECPATH} vspec.c

${MY8SPECPATH}/lp_man.o: ${UVA8PATH}/lp_man.c ${SPEC8HEADERS}
cd ${MY8SPECPATH}; \
$(I860CC) ${I_CFLAGS} -I${MY8SPECPATH} ${UVA8PATH}/lp_man.c

${MY8SPECPATH}/cube2.o: ${AFIT8PATH}/cube2.c ${AFIT8PATH}/cube2.c
                        ${SPEC8HEADERS} ${AFIT8_INC}/cube2.h
cd ${MY8SPECPATH}; \
$(I860CC) ${I_CFLAGS} ${AFIT8PATH}/cube2.c

${MY8SPECPATH}/u_null_filt.o: ${FILTER8PATH}/u_null_filt.c
                              ${SPEC8HEADERS}
cd ${MY8SPECPATH}; \
$(I860CC) ${I_CFLAGS} -I${MY8SPECPATH}
        ${FILTER8PATH}/u_null_filt.c

${MY8SPECPATH}/vfilt.o: ${FILTER8PATH}/vfilt.c ${SPEC8HEADERS}
cd ${MY8SPECPATH}; \
$(I860CC) ${I_CFLAGS} -DVHDL -I${MY8SPECPATH}
        ${FILTER8PATH}/vfilt.c
```

```
${I8_CKTPATH}/et_dff.o: et_dff.c ${I8_SIMPATH}/vsim.h
$(I860CC) ${I_CFLAGS} -I${I8_SIMPATH} et_dff.c

#                          -------- iPSC/2 ----------

ipsc: host node

host: ${MYSPECPATH}/host2.o
$(CC) -o host ${MYSPECPATH}/host2.o -host

node: ${NODE_OBJS}
$(CC) -o et_dff ${NODE_OBJS} -node

${MYSPECPATH}/host2.o: ${MYSPECPATH}/host2.c ${AFIT_INC}/cube2.h
cd ${MYSPECPATH}; \
$(CC) ${I_CFLAGS} -I${AFIT_INC} ${MYSPECPATH}/host2.c

${I_SIMPATH}/vsim.o: ${I_SIMPATH}/vsim.c ${I_SIMPATH}/vsim.h
                     ${MYSPECPATH}/application.h
cd ${I_SIMPATH}; \
$(CC) ${I_CFLAGS} -I${MYSPECPATH} vsim.c

${I_SIMPATH}/vinit.o: ${I_SIMPATH}/vinit.c ${I_SIMPATH}/vsim.h
                      ${MYSPECPATH}/application.h
cd ${I_SIMPATH}; \
$(CC) ${I_CFLAGS} -I${MYSPECPATH} vinit.c

${I_SIMPATH}/vtools.o: ${I_SIMPATH}/vtools.c ${I_SIMPATH}/vsim.h
cd ${I_SIMPATH}; \
$(CC) ${I_CFLAGS} vtools.c

${I_SIMPATH}/vspec.o: ${I_SIMPATH}/vspec.c ${I_SIMPATH}/vsim.h
                      ${MYSPECPATH}/application.h
cd ${I_SIMPATH}; \
$(CC) ${I_CFLAGS} -I${MYSPECPATH} vspec.c

${MYSPECPATH}/lp_man.o: ${UVAPATH}/lp_man.c ${SPECHEADERS}
cd ${MYSPECPATH}; \
$(CC) ${I_CFLAGS} -I${MYSPECPATH} ${UVAPATH}/lp_man.c

${MYSPECPATH}/cube2.o: ${AFITPATH}/cube2.c ${AFITPATH}/cube2.c
                       ${SPECHEADERS} ${AFIT_INC}/cube2.h
cd ${MYSPECPATH}; \
$(CC) ${I_CFLAGS} -I${AFIT_INC} -I${MYSPECPATH}
        ${AFITPATH}/cube2.c

${MYSPECPATH}/u_null_filt.o: ${FILTERPATH}/u_null_filt.c
                             ${SPECHEADERS}
cd ${MYSPECPATH}; \
$(CC) ${I_CFLAGS} -I${MYSPECPATH}
        ${FILTERPATH}/u_null_filt.c
```

```
${MYSPECPATH}/vfilt.o: ${FILTERPATH}/vfilt.c ${SPECHEADERS}
cd ${MYSPECPATH}; \
$(CC) ${I_CFLAGS} -DVHDL -I${MYSPECPATH}
        ${FILTERPATH}/vfilt.c

${I_CKTPATH}/et_dff.o: et_dff.c ${I_SIMPATH}/vsim.h
$(CC) ${I_CFLAGS} -I${I_SIMPATH} et_dff.c


#                          -------- SPARC ----------
vsim: ${S_OBJS}
$(CC) -o et_dff -g ${S_OBJS}

${S_SIMPATH}/vsim.o: ${S_SIMPATH}/vsim.c ${S_SIMPATH}/vsim.h
cd ${S_SIMPATH}; \
$(CC) ${S_CFLAGS} -I${S_SPECPATH} vsim.c

${S_SIMPATH}/vinit.o: ${S_SIMPATH}/vinit.c ${S_SIMPATH}/vsim.h
cd ${S_SIMPATH}; \
$(CC) ${S_CFLAGS} vinit.c

${S_SIMPATH}/vtools.o: ${S_SIMPATH}/vtools.c ${S_SIMPATH}/vsim.h
cd ${S_SIMPATH}; \
$(CC) ${S_CFLAGS} vtools.c

${S_CKTPATH}/et_dff.o: et_dff.c ${S_SIMPATH}/vsim.h
$(CC) ${S_CFLAGS} -I${S_SIMPATH} et_dff.c
```

After compiling, a sequential simulation may be run. For this example, the command is

```
et_dff > temp
```

The output, in **temp**, is already in time order[11]; however, **sgrep** sorts by time and then signal name. The following command is typed:

```
sgrep temp et_dff.out
```

The output is now sorted by time and signal name, and can be compared with Intermetrics' output for accuracy. Using **grep**, the values for **CKT_Q_OUT** are[12]

```
3 ns, CKT_Q_OUT from 0 to 1
6 ns, CKT_Q_OUT from 1 to 0
9 ns, CKT_Q_OUT from 0 to 1
12 ns, CKT_Q_OUT from 1 to 0
15 ns, CKT_Q_OUT from 0 to 1
```

[11] This is not the case for parallel simulations.

[12] For complete accuracy, *every* signal change should be examined. Only one signal was shown here for brevity.

```
18 ns, CKT_Q_OUT from 1 to 0
21 ns, CKT_Q_OUT from 0 to 1
24 ns, CKT_Q_OUT from 1 to 0
206 ns, CKT_Q_OUT from 0 to 1
359 ns, CKT_Q_OUT from 1 to 0
506 ns, CKT_Q_OUT from 0 to 1
```

B.3.6 Extracting Behavior Information using VMAP. Since **MAPPING** was defined, the output in **temp** also has behavioral information. Specifically, behavior names, id numbers, and dependencies, as shown here:

```
0 fs, executing beh 9: <<TBREEDEN>>ET_DFF_TEST_BENCH(STRUCTURAL)
Add behav 1 to active list at 0 fs
Add behav 2 to active list at 0 fs
Add behav 1 to active list at 15 ns
Add behav 2 to active list at 15 ns
Add behav 1 to active list at 20 ns
Add behav 2 to active list at 20 ns
Add behav 1 to active list at 50 ns
Add behav 2 to active list at 50 ns
Add behav 1 to active list at 150 ns
Add behav 2 to active list at 150 ns
Add behav 1 to active list at 200 ns
Add behav 2 to active list at 200 ns
Add behav 1 to active list at 300 ns
Add behav 2 to active list at 300 ns
Add behav 1 to active list at 350 ns
Add behav 2 to active list at 350 ns
Add behav 1 to active list at 450 ns
Add behav 2 to active list at 450 ns
Add behav 1 to active list at 500 ns
Add behav 2 to active list at 500 ns
0 fs, executing beh 8: <<TBREEDEN>>ET_DFF_TEST_BENCH(STRUCTURAL)
Add behav 3 to active list at 0 fs
Add behav 3 to active list at 100 ns
Add behav 3 to active list at 250 ns
Add behav 3 to active list at 400 ns
0 fs, executing beh 7: <<TBREEDEN>>ET_DFF(STRUCTURAL)
0 fs, executing beh 6: <<TBREEDEN>>ET_DFF(STRUCTURAL)
0 fs, executing beh 5: <<TBREEDEN>>NAND_GATE(SIMPLE)
Add behav 4 to active list at 3 ns
Add behav 7 to active list at 3 ns
0 fs, executing beh 4: <<TBREEDEN>>NAND_GATE(SIMPLE)
Add behav 5 to active list at 3 ns
Add behav 6 to active list at 3 ns
0 fs, executing beh 3: <<TBREEDEN>>NAND_GATE(SIMPLE)
Add behav 0 to active list at 3 ns
Add behav 2 to active list at 3 ns
0 fs, executing beh 2: <<TBREEDEN>>THREE_INPUT_NAND_GATE(SIMPLE)
```

```
Add behav 3 to active list at 3 ns
Add behav 5 to active list at 3 ns
0 fs, executing beh 1: <<TBREEDEN>>NAND_GATE(SIMPLE)
Add behav 0 to active list at 3 ns
Add behav 2 to active list at 3 ns
Add behav 4 to active list at 3 ns
0 fs, executing beh 0: <<TBREEDEN>>NAND_GATE(SIMPLE)
Add behav 1 to active list at 3 ns
```

Using **vmap**, this information can be filtered out of **temp** and saved. The **vmap** program attempts to "guess" the delays of each behavior, based on when dependent behaviors are scheduled. The user is given a chance to override these guesses. In most cases, the behaviors which represent gates show correct delays; the other "system" behaviors should be set to a delay of zero. Here is how **vmap** is used for this example:

```
neptune:~/et_dff>vmap temp et_dff.map
Collecting behavior names and delays...

ET_DFF_TEST_BENCH(STRUCTURAL) Delay = 0
ET_DFF(STRUCTURAL) Delay = 3000000
NAND_GATE(SIMPLE) Delay = 3000000
THREE_INPUT_NAND_GATE(SIMPLE) Delay = 3000000
Change delays? y

ET_DFF_TEST_BENCH(STRUCTURAL) Delay = 0
Change delay? n

ET_DFF(STRUCTURAL) Delay = 3000000
Change delay? y

Enter new delay: 0

NAND_GATE(SIMPLE) Delay = 3000000
Change delay? n

THREE_INPUT_NAND_GATE(SIMPLE) Delay = 3000000
Change delay? n

ET_DFF_TEST_BENCH(STRUCTURAL) Delay = 0
ET_DFF(STRUCTURAL) Delay = 0
NAND_GATE(SIMPLE) Delay = 3
THREE_INPUT_NAND_GATE(SIMPLE) Delay = 3000000
Change delays? n


Output written to et_dff.map

neptune:~/et_dff> more et_dff.map
9 ET_DFF_TEST_BENCH(STRUCTURAL) 0 1 2
```

```
8 ET_DFF_TEST_BENCH(STRUCTURAL) 0 3
7 ET_DFF(STRUCTURAL) 0
6 ET_DFF(STRUCTURAL) 0
5 NAND_GATE(SIMPLE) 3000000 4 7
4 NAND_GATE(SIMPLE) 3000000 5 6
3 NAND_GATE(SIMPLE) 3000000 0 2
2 THREE_INPUT_NAND_GATE(SIMPLE) 3000000 3 5
1 NAND_GATE(SIMPLE) 3000000 0 2 4
0 NAND_GATE(SIMPLE) 3000000 1
neptune:~/et_dff>
```

The format for **et_dff.out**, shown above, is

$$\{\text{behavior\_id behavior\_name delay } \{\text{dependent\_behaviors}\}_{0+} \text{ newline}\}_{1+}$$

Currently, the only way to map behavior numbers to behaviors is to compare the output of either VSIM or **vmap** to the schematic. For the edge-triggered D flip-flop, this is shown in Figure 65 on page 120.

B.3.7 Generating **.arcs** *and* **.map** *Files for Partitioning.*

B.3.7.1 A 1-LP Configuratio.n The whole circuit can be simulated as one LP. This configuration can be used to compare timing data, etc., with other configurations. An **lx1.map** file is not required; however, an **lp1.arcs** file is required and is written as so:

```
0
0
0
0
0
```

B.3.7.2 A 2-LP Confguration. The first configuration to be tested has 2 LPs. LP0 contains behaviors 0, 1, 2, 3, 8, and 9. LP1 contains behaviors 4, 5, 6, and 7. See Figure 66 on page 121. The arcs file that SPECTRUM uses is **lp2.arcs**, and it contains the following mapping:

```
0
0
0
1
1
1
1
3000000

1
1
```

```
0
0
0
1
0
0
```

The map file is for VSIM to identify which LPs "own" which behaviors. VSIM always expects this filename to be "`lpx.map`", where the number of LPs replaces `x`. Therefore, `lp2.map` is written as follows:

```
0 0
1 0
2 0
3 0
4 1
5 1
6 1
7 1
8 0
9 0
```

B.3.7.3 A 3-LP Configuration with Feedback. This configuration, shown in Figure 67 (page 121), is used to demonstrate VSIMs capability to handle feedback among LPs. The `.arcs` file is `lp3.arcs`, and contains the following:

```
0
0
0
2
1 2
2
1 2
3000000 3000000

1
2
0 2
0 0
0 0
2
0 2
1
2
1
2
3000000
```

```
2
2
0 1
0 0
0 0
2
0 1
1
1
1
1
3000000
```

Then, `lp3.map` is written as follows:

```
0 0
1 0
2 0
3 0
4 1
5 2
6 1
7 2
8 0
9 0
```

B.3.8 Parallel Simulation. Prior to simulating in parallel, `MAPPING` is turned off in `vsim.h`. This is not a requirement, but mapping information is no longer needed. Prior to changing the number of LPs for any simulation, `application.h` is modified, using `setlps`, to define `NUM_PROCS` and `INPUT_ARCS`, the number of LPs and the `.arcs` filename, respectively. The intermediate C code, its header files, and the `lpx.arcs` and `lpx.map` files are sent to the hypercube and compiled each time the number of LPs is changed.

B.3.8.1 Simulating the Edge-Triggered D Flip-Flop as one LP. The number of LPs is set to 1 and the same makefile is used (this time typing "`make`") to compile. The simulation is started by typing `host`. Here is an example:

```
c386 8:host
Which application do you want to use?:et_dff
Enter the command line arguments for the program
>
Is assignment of logical processes to nodes to be from a file? (y/n) -> n
How many cube nodes do you want to use?:1
How many LP's are in this application?:1
Do you want to use the 'natural' node assignment? (y/n): y
```

```
Getting cube of size 1 - stand by.
load -H -p 0 0 et_dff
startcube
Cube Loaded
LAST_TIME message from LP 0 on node 0, pid 0.

End stats messages:
LP 0 (node 0, pid 0):   0 received,     0 sent.
Max message count set at 10, Max messages removed was 0.
HOST: Total CPU time waiting: 0.000000 (msecs)
HOST: Wall clock time loading cube: 7 (secs)
HOST: Wall clock time waiting: 4 (secs)
c386 9:
```

Now, the output is found in `lp1.out` and can be compared to `etdff.out`, which the previously verified output. Also, an LP report file, `log0` is generated by SPECTRUM with the following information:

```
LP 0 wall time taken is 4.194 (secs)
LP 0 messages received 0
LP 0 messages sent 0
```

B.3.8.2 Simulating the Edge-Triggered D Flip-Flop as more than one LP. The process is the same as for one LP; however, the output is combined in the `lpx.out` files. For example, the two LP configuration's output is found in `lp0.out` and `lp1.out`. These two files are concatenated and `sgrep` is used to sort them. The output from `sgrep` is verified against `et_dff.out`. The results of all 1, 2, and 3 LP configurations are shown in Table 8.[13][14]

B.3.9 Summary. This guide demonstrates how to compile a VHDL circuit with the Intermetrics VHDL toolset, intercept the intermediate C code, and compile and link with AFIT's parallel VHDL simulator (VSIM). VSIM simulations of an edge-triggered D flip-flop are demonstrated for a single processor and in parallel on the Intel iPSC/2 Hypercube.

[13] These results were run one time for each configuration, and are for comparison purposes. If statistics are required, more runs would have to be made.

[14] For the 3 LP/2 node configuration, LP0 was loaded on node 0, and LPs 1 and 2 were loaded on node 1.

THIS PAGE IS MISSING IN ORIGINAL DOCUMENT

page 114

```
-----------------------------------------------

entity NAND_GATE is
  generic (gate_delay: TIME := 3 ns);
  port (IN_1,IN_2: in BIT := '0';
        OUT_1: out BIT := '0');
end NAND_GATE;

-----------------------------------------------

architecture SIMPLE of NAND_GATE is

begin
  OUT_1 <= IN_1 nand IN_2 after gate_delay ;
end SIMPLE ;

-----------------------------------------------
-----------------------------------------------

entity THREE_INPUT_NAND_GATE is
  generic (gate_delay: TIME := 3 ns);
  port (IN_1,IN_2,IN_3: in BIT := '0';
        OUT_1: out BIT := '0');
end THREE_INPUT_NAND_GATE;

-----------------------------------------------

architecture SIMPLE of THREE_INPUT_NAND_GATE is

begin
  OUT_1 <= not (IN_1 and IN_2 and IN_3)
           after gate_delay ;
end SIMPLE ;

-----------------------------------------------
```

Figure 58. VHDL Descriptions of Two- and Three-Input NAND Gates

```
-----------------------------------------------
-- Lt T. Andy Breeden, GCE-92D, 4 Aug 92
-- Edge-Triggered D Flip-Flop (structural)
-----------------------------------------------

entity ET_DFF is

   port (D,CP: in Bit;
         Q: out Bit;
         Q_Bar: out Bit);
begin
end ET_DFF;

-----------------------------------------------
architecture Structural of ET_DFF is

   component A_NAND_Gate
      port (In_1, In_2: in Bit;
            Out_1: out Bit);
   end component;

   component A_3Input_NAND_Gate
      port (In_1, In_2, In_3: in Bit; Out_1: Out Bit);
   end component;

   signal X1_Out, X2_Out, X3_Out, X4_Out: Bit;
   signal X5_Out, X6_Out: Bit;

begin
   X1: A_NAND_Gate port map (X4_Out,X2_Out,X1_Out);
   X2: A_NAND_Gate port map (X1_Out,CP,X2_Out);
   X3: A_3Input_NAND_Gate port map (X2_Out,CP,X4_Out,X3_Out);
   X4: A_NAND_Gate port map (X3_Out,D,X4_Out);
   X5: A_NAND_Gate port map (X2_Out,X6_Out,X5_Out);
   X6: A_NAND_Gate port map (X5_Out,X3_Out,X6_Out);
   Q <= X5_Out;
   Q_Bar <= X6_Out;
end Structural;

-----------------------------------------------
```

Figure 59. Structural VHDL Description of Edge-triggered D Flip-flop

```
-----------------------------------------
-- Test Bench for Edge-Triggered D Flip-Flop
-- Lt T. Andy Breeden, GCE-92D, 4 Aug 92
-----------------------------------------

entity ET_DFF_Test_Bench is
end ET_DFF_Test_Bench;

architecture Structural of ET_DFF_Test_Bench is

   component Test_Circuit
          port (D,CP: in Bit;
                Q,Q_Bar: out Bit);
   end component;

   signal a,b,Ckt_Q_Out,Ckt_Q_Bar_Out: Bit;

begin

   Circuit: Test_Circuit port map (a, b,
                                   Ckt_Q_Out, Ckt_Q_Bar_Out);

   a <= '0' after 0 ns, '1' after 100 ns,
        '0' after 250 ns, '1' after 400 ns;

   b <= '0' after 0 ns, '1' after 15 ns, '0' after 20 ns,
        '1' after 50 ns, '0' after 150 ns, '1' after 200 ns,
        '0' after 300 ns, '1' after 350 ns, '0' after 450 ns,
        '1' after 500 ns;


end Structural;
```

Figure 60. VHDL Description of Test Bench for Edge-triggered D Flip-flop

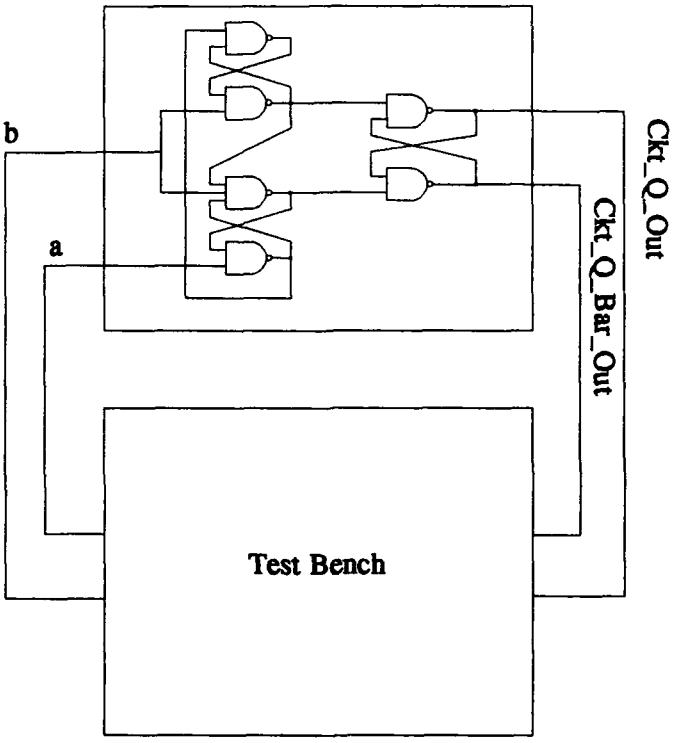


Figure 61. Schematic of Test Bench for Edge-triggered D Flip-flop

```
-----------------------------------------
-- Configuration file to connect Edge-Triggered
--  DFF to test bench.
--
--
-- Lt T. Andy Breeden, GCE-92D, 4 Aug 92
-----------------------------------------

Library work;
use work.all;
configuration ET_DFF_Config of ET_DFF_Test_Bench is
for Structural
   for Circuit: Test_Circuit
      use entity work.ET_DFF(Structural);
      for Structural
         for all: A_NAND_Gate
            use entity work.NAND_GATE(Simple);
         end for;
         for all: A_3Input_NAND_Gate
            use entity work.Three_Input_NAND_Gate(Simple);
         end for;
      end for;
   end for;
end for;
end ET_DFF_Config;
```

Figure 62. VHDL Description of Configuration File for Edge-triggered D Flip-flop

```
-----------------------------------------------
-- Output for Edge-Triggered DFF simulation using
-- Intermetrics' Report Control Language (RCL)
--
-- Lt T. Andy Breeden, GCE-92-D, 4 Aug 92
-----------------------------------------------

simulation_report ET_DFF_Sim is
begin
   select_signal: a,b,Ckt_Q_Out,Ckt_Q_Bar_Out;
   sample_signals by_event in ns;
end;
```

Figure 63. VHDL Report Description for Edge-triggered D Flip-Flop

```
#!/bin/csh -v
vhdl ~/vhdl/aox_gates/nand_nor
vhdl et_dff.vhd
vhdl et_dff_test_bench
vhdl et_dff_config
mg '-debug=cknd nand_gate(simple)'
mg '-debug=cknd three_input_nand_gate(simple)'
mg '-debug=cknd et_dff(structural)'
mg '-debug=cknd et_dff_test_bench(structural)'
mg '-debug=cknd -top et_dff_config'
build '-debug=cknd -replace -ker=et_dff et_dff_config'
sim et_dff
rg et_dff et_dff.rcl
```

Figure 64. Shell Script for Compiling, Model Generating, Building, and Simulating the Edge-triggered D Flip-flop using Intermetrics' Simulator

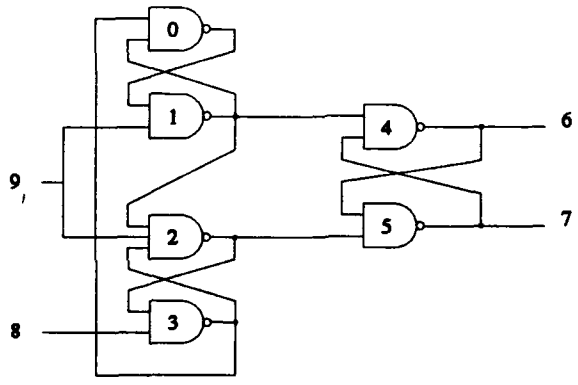


Figure 65. Edge-Triggered D Flip-flop Labeled with Behavior Id Numbers

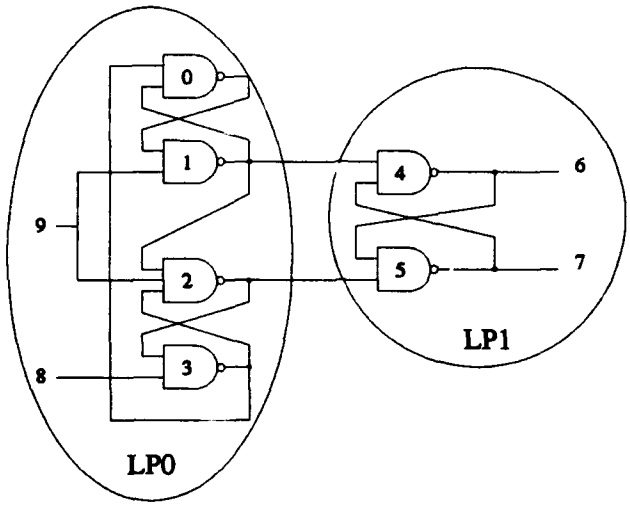


Figure 66. Edge-Triggered D Flip-flop Partitioned Into 2 LPs

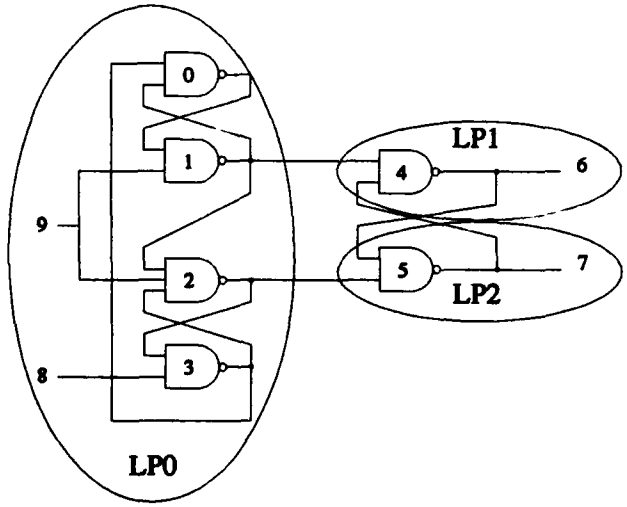


Figure 67. Edge-Triggered D Flip-flop Partitioned Into 3 LPs

Appendix C. *Subset of VHDL Source Code for Parallel Simulation*

The subset of circuits that can be simulated with VSIM includes hierarchical structural descriptions of logic gates. This appendix discusses the subset and syntax for logic gates, structural connections, the test bench, and configurations.

C.1 Logic Gates.

Logic gates are designed as entity/architecture pairs. Input and output signals for logic gates must be of type **Bit**. The number of inputs and outputs is not restricted. Default values may be assigned. Gate delays are of type **time**, and may be constants or generics. Processes may use **wait** statements only if they wait on *all* inputs. Logical operators **and**, **or**, **nand**, **nor**, and **xor** may be used. The adding operator (+) may be used to add values of type **time**. Here are some examples of acceptable logic gate descriptions:

```
entity AND_GATE is
  generic (gate_delay: TIME := 3 ns);
  port (IN_1,IN_2: in BIT := '0';
        OUT_1: out BIT := '0');
end AND_GATE;

architecture SIMPLE of AND_GATE is

begin
  OUT_1 <= IN_1 and IN_2 after gate_delay ;
end SIMPLE ;
```

```
Entity THREE_INPUT_AND is
    Port (in_1, in_2, in_3 : in BIT := '0'; out_1 : out BIT := '0');
    Constant Delay : Time := 5 ns;
end THREE_INPUT_AND;

Architecture BEHAV_3AND of THREE_INPUT_AND is

begin
  process begin
```

```
    OUT_1 <= IN_1 and IN_2 and IN_3 after delay;
    wait on IN_1, IN_2, IN_3;
  end process;
end BEHAV_3AND;
```

```
entity GND_BOX is
   Port (GZ : Out Bit );
end GND_BOX;

architecture BEHAVIORAL of GND_BOX is

begin
   GZ <= '0';
end BEHAVIORAL;
```

C.2 Structural Connection of Logic Gates.

Circuits are built hierarchically in entity/architecture pairs by structurally connecting logic gate components or other structural descriptions. Assertions can be raised at this point. An assertion of type **error** or **fatal** will abort the simulation. Component port maps use either named or positional notation for signal assignments. Bit vectors may also be used. Here is an example of an SR flip-flop that structurally connects two nor gates:

```
entity SRFF is

   port (S,R: in Bit;
         Q: out Bit;
         Q_Bar: out Bit);
begin
   SRFF_Constraint_Check:
      assert not (S='1' and R='1')
         report "Both S and R equal to '1'"
            severity Error;
end SRFF;

architecture Structural of SRFF is

   component A_NOR_Gate
```

```
      port (In_1, In_2: in Bit; Out_1: out Bit);
   end component;

   signal Q_Bar_In: Bit;
   signal Q_In: Bit;

begin
   X1: A_NOR_Gate port map (R,Q_Bar_In,Q_In);
   X2: A_NOR_Gate port map (Q_In,S,Q_Bar_In);
   Q <= Q_In;
   Q_Bar <= Q_Bar_In;
end Structural;
```

This carry save adder shows positional notation, use of bit vectors, and structural connections of both gates (inverters) and full adders which are structurally defined elsewhere:

```
entity CSA8 is
      Port (       A : In    Bit_VECTOR (7 downto 0) := "00000000";
                   B : In    Bit_VECTOR (7 downto 0) := "00000000";
                   C : In    Bit_VECTOR (7 downto 0) := "00000000";
             HI_CSA_BIT : In    Bit := '0';
             LO_CSA_BIT : In    Bit := '0';
               CARRY : Out   Bit_VECTOR (7 downto 0) := "00000000";
             HI_SUM_BIT : Out   Bit := '0';
             LO_SUM_BIT : Out   Bit := '0';
                 SUM : Out   Bit_VECTOR (7 downto 0) := "00000000" );
end CSA8;

architecture SCHEMATIC of CSA8 is

   signal   N_1 : Bit;
   signal   N_2 : Bit;

   component INV_1
      Port (In_1 : In    Bit := '0';
            Out_1 : Out   Bit := '0' );
   end component;

   component FULL_ADDER
      Port (AIN : In Bit := '0';
            BIN : In Bit := '0';
            CIN : In Bit := '0';
            CARRY : Out Bit := '0';
            SUM : Out Bit := '0' );
   end component;
```

```
begin

   I_9 : INV_1
      Port Map ( In_1=>N_2, Out_1=>LO_SUM_BIT );
   I_10 : INV_1
      Port Map ( In_1=>A(0), Out_1=>N_2 );
   I_11 : INV_1
      Port Map ( In_1=>N_1, Out_1=>HI_SUM_BIT );
   I_12 : INV_1
      Port Map ( In_1=>C(7), Out_1=>N_1 );
   I_1 : FULL_ADDER
      Port Map ( AIN=>HI_CSA_BIT, BIN=>B(7), CIN=>C(6), CARRY=>CARRY(7),
                 SUM=>SUM(7) );
   I_2 : FULL_ADDER
      Port Map ( AIN=>A(7), BIN=>B(6), CIN=>C(5), CARRY=>CARRY(6),
                 SUM=>SUM(6) );
   I_3 : FULL_ADDER
      Port Map ( AIN=>A(6), BIN=>B(5), CIN=>C(4), CARRY=>CARRY(5),
                 SUM=>SUM(5) );
   I_4 : FULL_ADDER
      Port Map ( AIN=>A(5), BIN=>B(4), CIN=>C(3), CARRY=>CARRY(4),
                 SUM=>SUM(4) );
   I_5 : FULL_ADDER
      Port Map ( AIN=>A(4), BIN=>B(3), CIN=>C(2), CARRY=>CARRY(3),
                 SUM=>SUM(3) );
   I_6 : FULL_ADDER
      Port Map ( AIN=>A(3), BIN=>B(2), CIN=>C(1), CARRY=>CARRY(2),
                 SUM=>SUM(2) );
   I_7 : FULL_ADDER
      Port Map ( AIN=>A(2), BIN=>B(1), CIN=>C(0), CARRY=>CARRY(1),
                 SUM=>SUM(1) );
   I_8 : FULL_ADDER
      Port Map ( AIN=>A(1), BIN=>B(0), CIN=>LO_CSA_BIT, CARRY=>CARRY(0),
                 SUM=>SUM(0) );
end SCHEMATIC;
```

C.3 Test Bench and Input Vectors.

Test benches are used to connect the circuit under test to a series of input test signals. The inputs may be of type `bit` or `bit_vector`; however, each bit of a bit vector must be assigned values individually. VSIM terminates after 2000 ns; therefore, ***no input signal should be assigned a value beyond 2000 ns.*** Here is an example of a test bench for the 16-bit bit/byte shifter:

```
entity Shifter_TB is
end Shifter_TB;

architecture Structural of Shifter_TB is

   component Test_Circuit
      Port ( SHIFTER_CONTROL : In Bit_Vector (2 downto 0);
             SHIFTER_INPUT : In Bit_Vector (15 downto 0);
             SHIFTER_OUTPUT : Out Bit_Vector (15 downto 0) );
   end component;

   signal Control: Bit_Vector(2 downto 0);
   signal Input: Bit_Vector(15 downto 0);
   signal Output: Bit_Vector(15 downto 0);

begin

   Circuit: Test_Circuit port map (Control, Input, Output);

   -- Use Input = 0101010101010101 after 10 ns, then
   --             0000111100001111 after 250 ns,

   Input(0)  <= '1' after 10 ns, '1' after 250 ns;
   Input(1)  <= '0' after 10 ns, '1' after 250 ns;
   Input(2)  <= '1' after 10 ns, '1' after 250 ns;
   Input(3)  <= '0' after 10 ns, '1' after 250 ns;
   Input(4)  <= '1' after 10 ns, '0' after 250 ns;
   Input(5)  <= '0' after 10 ns, '0' after 250 ns;
   Input(6)  <= '1' after 10 ns, '0' after 250 ns;
   Input(7)  <= '0' after 10 ns, '0' after 250 ns;
   Input(8)  <= '1' after 10 ns, '1' after 250 ns;
   Input(9)  <= '0' after 10 ns, '1' after 250 ns;
   Input(10) <= '1' after 10 ns, '1' after 250 ns;
   Input(11) <= '0' after 10 ns, '1' after 250 ns;
   Input(12) <= '1' after 10 ns, '0' after 250 ns;
   Input(13) <= '0' after 10 ns, '0' after 250 ns;
   Input(14) <= '1' after 10 ns, '0' after 250 ns;
   Input(15) <= '0' after 10 ns, '0' after 250 ns;

   --- Check      left shift,        right shift,
   ---            left shift 8,      right shift 8,    pass
   Control(0) <= '1' after 20 ns,   '0' after 50 ns,
                 '1' after 100 ns,  ' after 150 ns,  '0' after 200 ns,
                 '1' after 300 ns,  '0' after 350 ns,
                 '1' after 400 ns,  ' after 450 ns,  '0' after 500 ns;
   Control(1) <= '0' after 20 ns,   '1' after 50 ns,
                 '1' after 100 ns,  ' after 150 ns,  '0' after 200 ns,
                 '0' after 300 ns,  '1' after 350 ns,
                 '1' after 400 ns,  ' after 450 ns,  '0' after 500 ns;
   Control(2) <= '0' after 20 ns,   '0' after 50 ns,
                 '0' after 100 ns,  ' after 150 ns,  '0' after 200 ns,
```

```
                '0' after 300 ns,  '0' after 350 ns,
                '0' after 400 ns,  ' after 450 ns,  '0' after 500 ns;

end Structural;
```

C.4 Configuration Descriptions.

Configuration specifications are used to bind component instances to design entities. Configurations may either be assigned all at once at the top level, or at each intermediate step in hierarchical fashion. The latter saves a great deal of file space with respect to the intermediate C code; this increases the chances that large circuits will compile on the hypercubes without running out of memory.

The following is an example of a single top-level configuration for the carry lookahead adder:

```
use WORK.TEST_CL_ADDER;

Configuration S_CONF_CLA of TEST_CL_ADDER is
   for INSTANTIATE_CL_ADDER
      for CLA : CARRY_LOOKAHEAD_ADDER
         use Entity WORK.CARRY_LOOKAHEAD_ADDER(STRUCT_CLA);
            for STRUCT_CLA
               for all : AND_GATE
                  use Entity WORK.AND_GATE(SIMPLE);
               end for;
               for all : THREE_INPUT_AND
                  use Entity WORK.THREE_INPUT_AND(BEHAV_3AND);
               end for;
               for all : FOUR_INPUT_AND
                  use Entity WORK.FOUR_INPUT_AND(BEHAV_4AND);
               end for;
               for all : FIVE_INPUT_AND
                  use Entity WORK.FIVE_INPUT_AND(BEHAV_5AND);
               end for;
               for all : OR_GATE
                  use Entity WORK.OR_GATE(SIMPLE);
               end for;
               for all : THREE_INPUT_OR
                  use Entity WORK.THREE_INPUT_OR(BEHAV_3OR);
               end for;
```

```
            for all : FOUR_INPUT_OR
               use Entity WORK.FOUR_INPUT_OR(BEHAV_4OR);
            end for;
            for all : FIVE_INPUT_OR
               use Entity WORK.FIVE_INPUT_OR(BEHAV_5OR);
            end for;
            for all : XOR_GATE
               use Entity WORK.XOR_GATE(SIMPLE);
            end for;
         end for;
      end for;
   end for;
end S_CONF_CLA;
```

The wallace tree is an example of using hierarchical configurations. First, full adders are configured with logic gates, then carry save adders are configured with full adders (and more inverters), etc. While this method has the benefit of smaller intermediate C code, the postprocessor output must be modified as explained on page 95 of Appendix B. Here are the wallace tree configuration descriptions:

```
configuration CFG_FULL_ADDER of Work.FULL_ADDER is

   for SCHEMATIC
      for I_1, I_2, INV_1CARRY, INV_1A, INV_1B, INV_1C: INV_1
         use entity WORK.Inv(Simple);
      end for;
      for NANDBC, NANDCARRY, NANDAC, NANDSUM, NANDAB: NAND_2
         use entity WORK.Nand_Gate(Simple);
      end for;
      for NAND3CARRY, NAND3OR, NAND3ABC: NAND_3
         use entity WORK.Three_Input_Nand_Gate(Simple);
      end for;
   end for;

end CFG_FULL_ADDER;

configuration CFG_CSA8 of Work.CSA8 is

   for SCHEMATIC
      for I_9, I_10, I_11, I_12: INV_1
         use entity WORK.Inv(Simple);
      end for;
```

```
      for I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8: FULL_ADDER
         use configuration WORK.CFG_FULL_ADDER;
      end for;
   end for;

end CFG_CSA8;

configuration CFG_WALLACE_TREE_1 of Work.WALLACE_TREE_1 is

   for SCHEMATIC
      for I_17, I_18, I_19, I_20, I_21, I_22, I_23: GND_BOX
         use entity WORK.Gnd_Box(Behavioral);
      end for;
      for I_11, I_12: FULL_ADDER
         use configuration WORK.CFG_FULL_ADDER;
      end for;
      for I_15, I_16, I_13, I_14, I_8, I_9: INV_1
         use entity WORK.Inv(Simple);
      end for;
      for I_6, I_4, I_5, I_2, I_3: CSA8
         use configuration WORK.CFG_CSA8;
      end for;
      for I_1: MCAND_GEN
         use configuration WORK.CFG_MCAND_GEN;
      end for;
   end for;

end CFG_WALLACE_TREE_1;

configuration CFG_WALLACE_TREE_2 of Work.WALLACE_TREE_2 is

   for SCHEMATIC
      for I_26, I_27, I_25: GND_BOX
         use entity WORK.GND_BOX(Behavioral);
      end for;
      for I_21, I_22: INV_1
         use entity WORK.Inv(Simple);
      end for;
      for I_23, I_24, I_20, I_9, I_10, I_11, I_6, I_12, I_13, I_14, I_15,
          I_16, I_17, I_18, I_19, I_1, I_2, I_3, I_4, I_5: FULL_ADDER
         use configuration WORK.CFG_FULL_ADDER;
      end for;
      for I_8: WALLACE_TREE_1
         use configuration WORK.CFG_WALLACE_TREE_1;
      end for;
   end for;

end CFG_WALLACE_TREE_2;

configuration CGF_Wallace_TB of Work.Wallace_TB is
```

```
   for Structural
      for Circuit: Test_Circuit
         use configuration work.CFG_WALLACE_TREE_2;
      end for;
   end for;

end CGF_Wallace_TB;
```

Appendix D. *Design of the Wallace Tree Multiplier*

The wallace tree multiplier is the largest circuit simulated with VSIM on the Intel Hypercubes. It is created and verified in MVL-7 logic using Synopsis design tools. For AFIT VSIM simulations, MVL-7 bits and bit vectors are changed to type `bit` and `bit_vector`.

The hierarchical design of the multiplier has two advantages. First, the corresponding intermediate C code from Intermetrics' compiler is smaller than intermediate C code for an equivalent large, flat circuit description. Second, breaking the multiplier into hierarchical components provides logical, concurrent subcomponents that may be partitioned among the nodes of a parallel computer.

The design is taken from Hwang and Briggs (19). Figure 68 shows the multiplier as a tree of carry save adders followed by a carry propagate adder. Two eight bit numbers, A and B, are fed into a multiplicand generator which generates intermediate results and shifts them accordingly. These results go through the series of carry save adders, and then the carry propagate adder where the twelve bit product, P, is generated.

The VHDL hierarchy is shown in Figure 69. The overall circuit, wallace_tree_2, consists of wallace_tree_1 and a set of full adders that make the carry propagate adder. The wallace_tree_1 description includes the multiplicand generator and the carry save adders. In turn, the carry save adders are made with full adders. All full adders are composed of nand gates and inverters.

The schematics for all components are as follows: wallace_tree_2 is shown in Figure 70, wallace_tree_1 is shown in Figure 71, the multiplicand generator and two "subgenerators" are shown in Figures 72 and 73, the carry save adder is shown in Figure 74, and the full adder is shown in Figure 75.

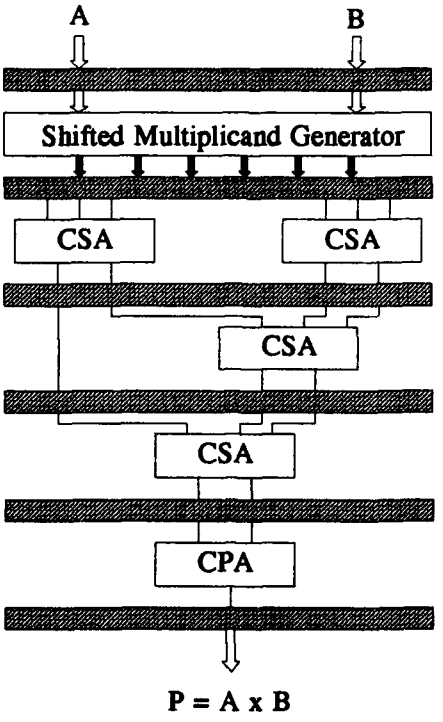


Figure 68. Wallace Tree Multiplier

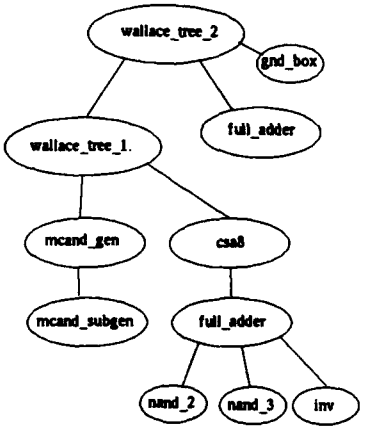


Figure 69. Hierarchy of VHDL Source Code for the Wallace Tree Multiplier

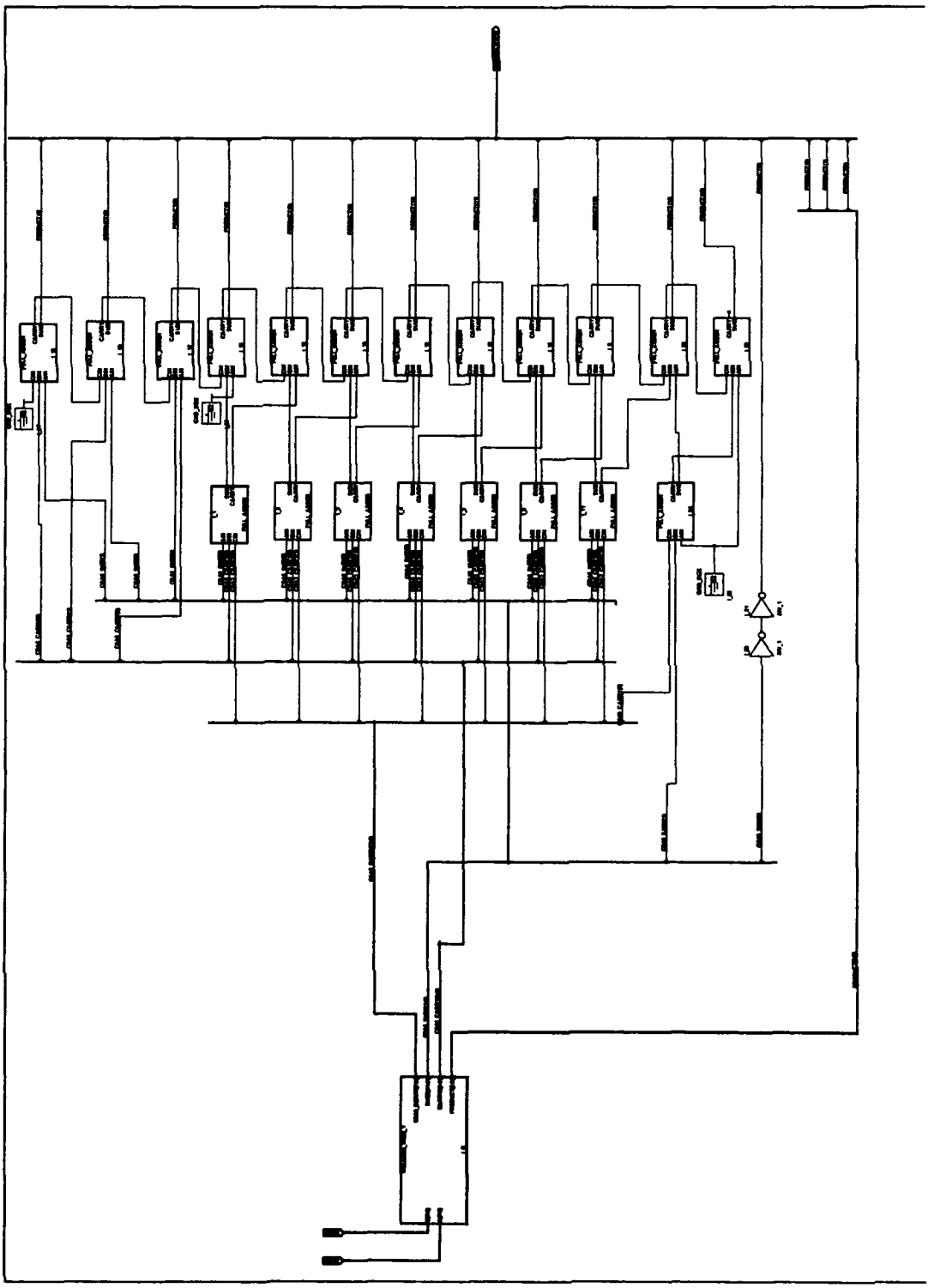


Figure 70. Top Level Schematic of Wallace Tree (wallace_tree_2)

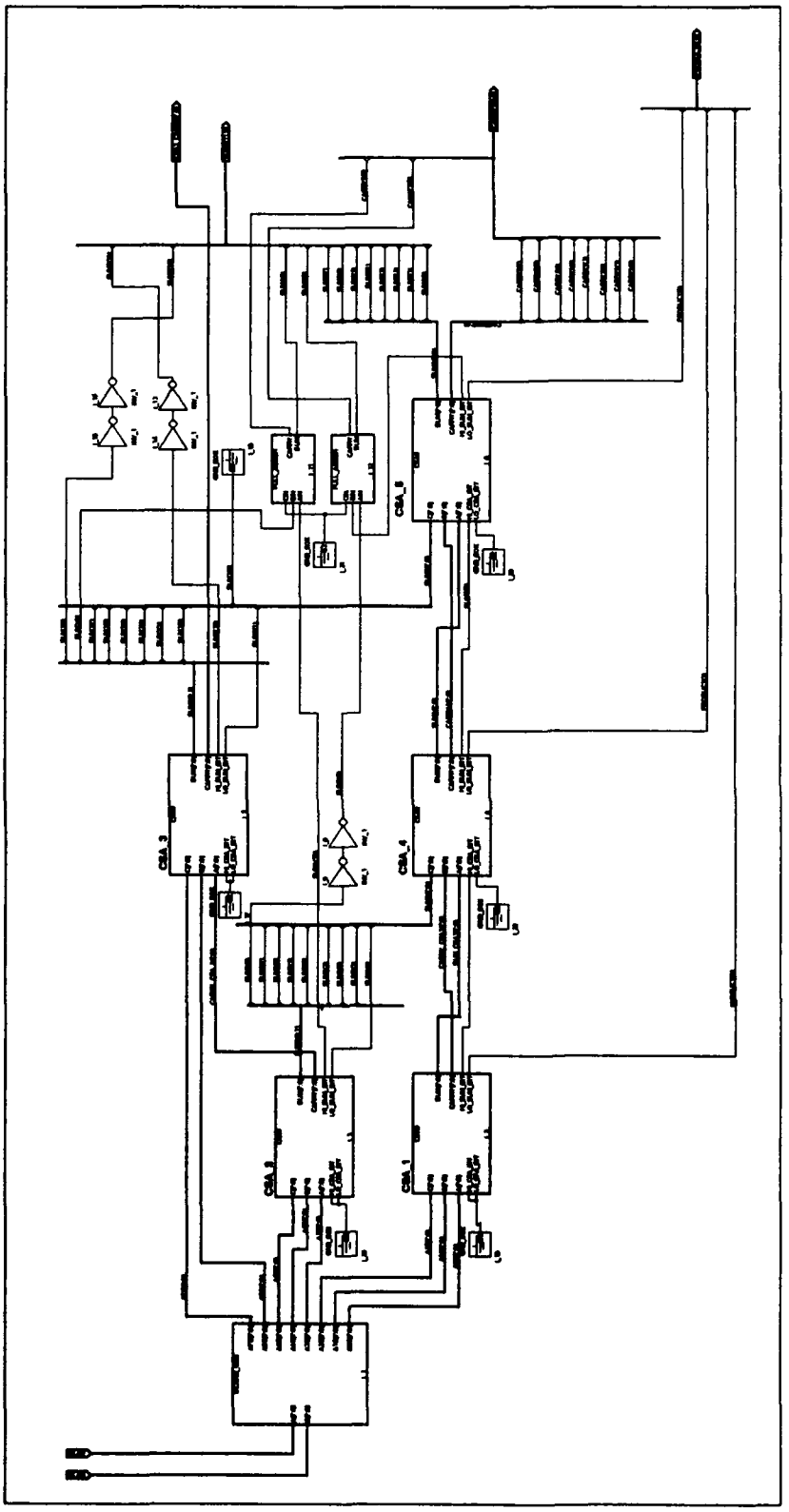


Figure 71. Schematic of Carry Save Adder Tree (wallace_tree_1)

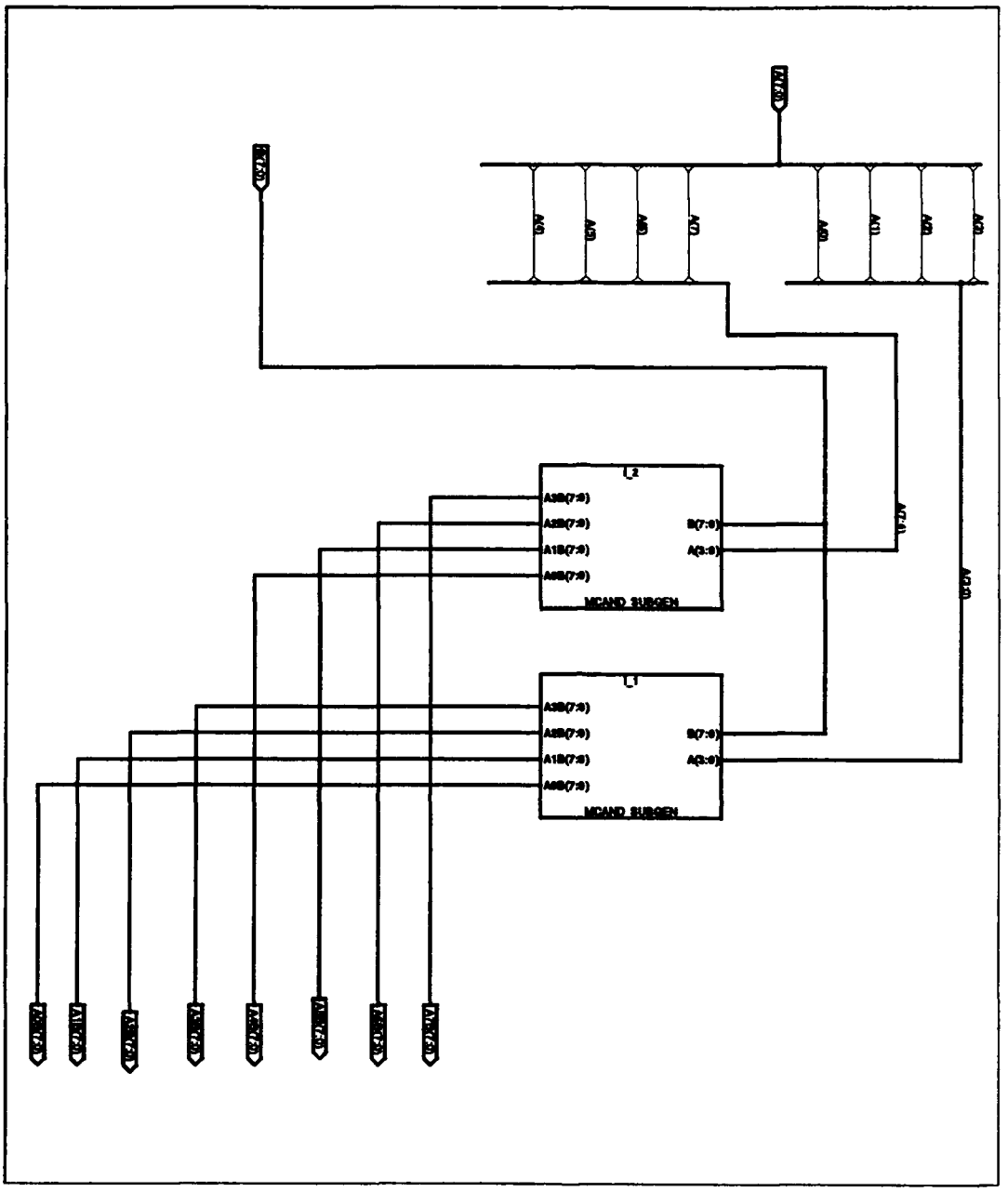


Figure 72. Multiplicand Generator

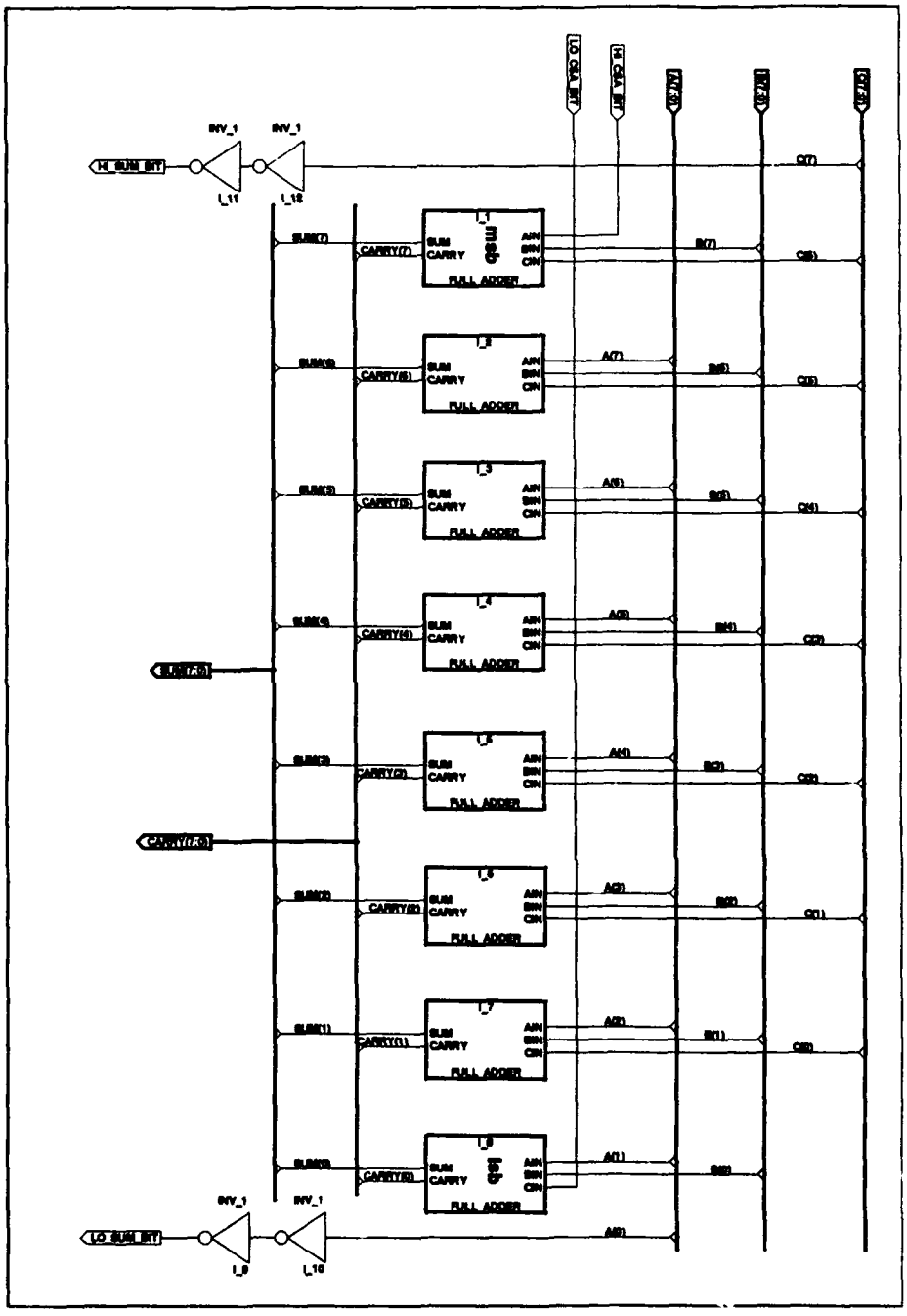


Figure 73. Multiplicand Subgenerator

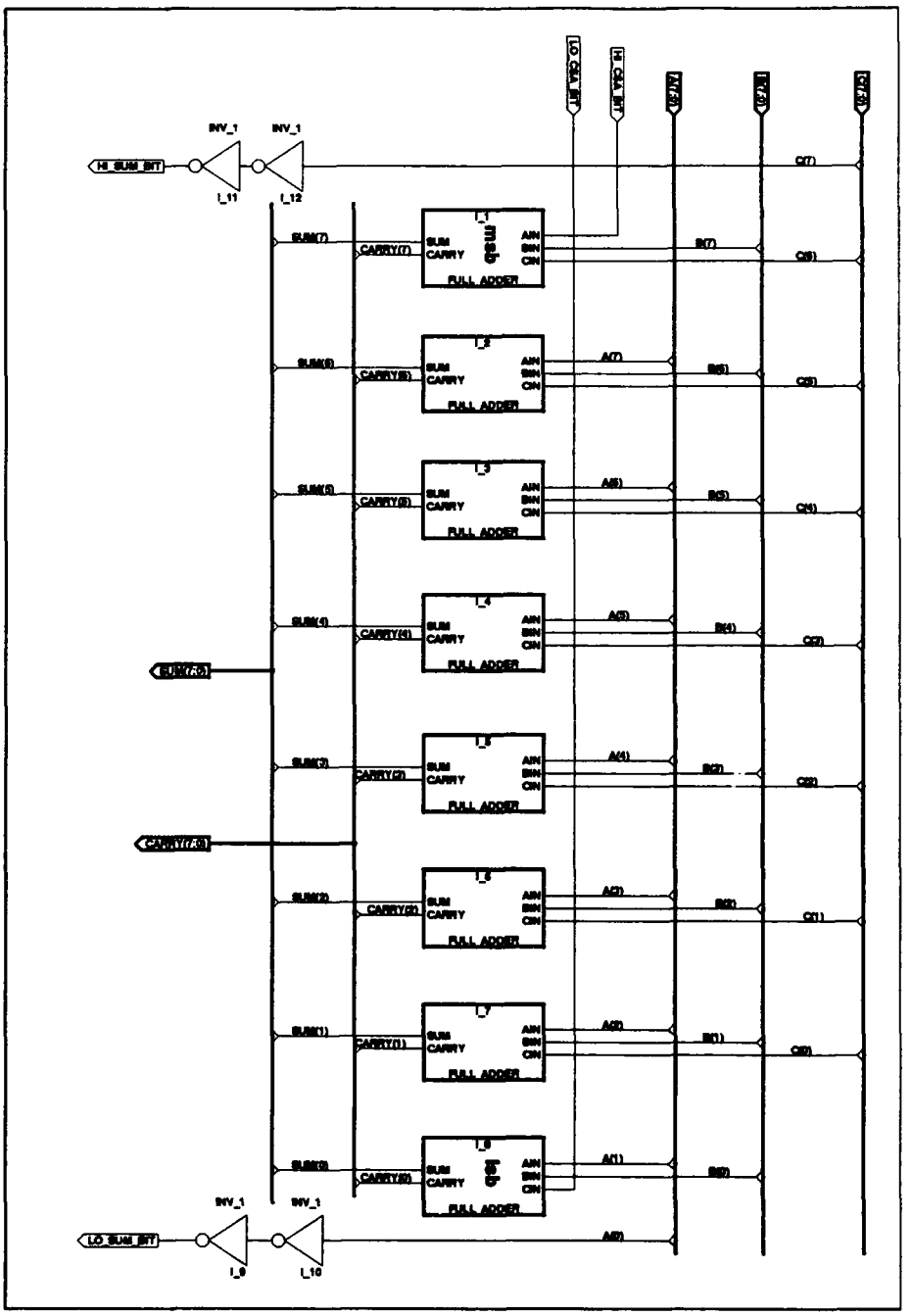


Figure 74. Carry Save Adder used in Wallace Tree Multiplier

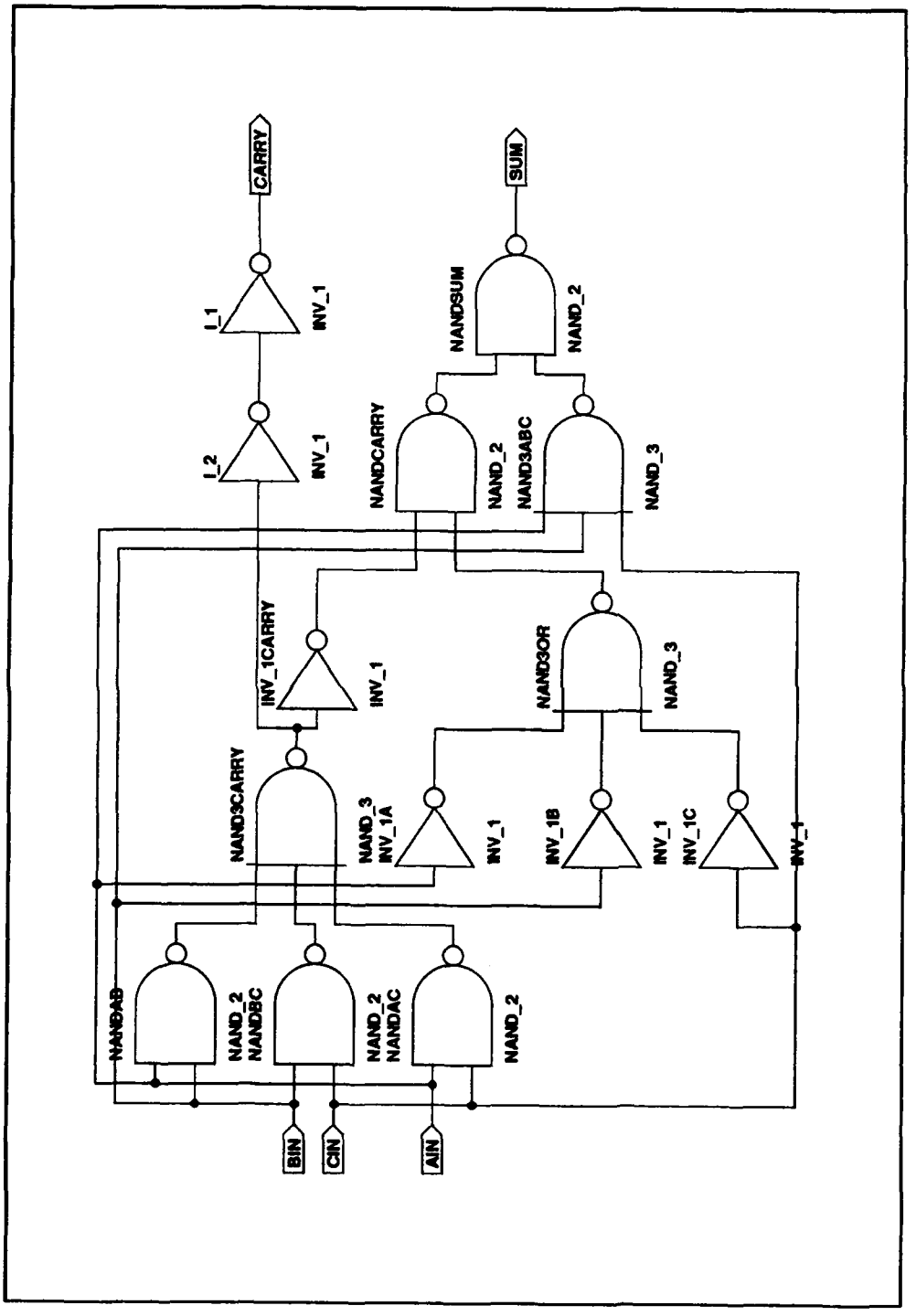


Figure 75. Full Adder used in Wallace Tree Multiplier

Appendix E. *Summary of Performance Data*

Each simulation configuration is summarized in Tables 9 and 10. The times reported correspond to the average execution time of 30 simulations per configuration. The reported speedups correspond to the simulation time of the slowest LP, neglecting the overhead of initializing and closing each process. For example, if a two-LP simulation is run and LP0 reports a time of $50ns$ while LP1 reports a time of $53ns$, the time for the simulation is considered to be $53ns$. Speedups are related to one LP simulations of otherwise identical configurations.

Table 9. Summary of Performance Data

Circuit	Hypercube	LPs	Nodes	Input Vectors	Time (ms)	Std Dev	Min	Max	Speedup
carry save adder	iPSC/2	1	1	20	1682	0.466	1681	1682	1.0
carry save adder	iPSC/2	2	2	20	515	0.183	515	516	3.3
carry save adder	iPSC/2	4	4	20	182	0.480	182	183	9.2
carry save adder	iPSC/2	8	8	20	81	0.450	80	81	20.8
carry save adder	iPSC/860	1	1	20	261	0.365	260	262	1.0
carry save adder	iPSC/860	2	2	20	62	0.450	61	62	4.2
carry save adder	iPSC/860	4	4	20	21	0.466	21	22	12.3
carry save adder	iPSC/860	8	8	20	9	0.466	9	10	28.1
carry propagate adder	iPSC/2	1	1	20	1101	0.183	1101	1102	1.0
carry propagate adder	iPSC/2	2	2	20	469	15.699	436	512	2.3
carry propagate adder	iPSC/2	4	4	20	468	97.798	412	892	2.4
carry propagate adder	iPSC/2	8	8	20	699	40.344	620	791	1.6
carry propagate adder	iPSC/860	1	1	20	151	0.498	151	152	1.0
carry propagate adder	iPSC/860	2	2	20	124	15.062	66	147	1.2
carry propagate adder	iPSC/860	4	4	20	293	20.146	250	339	0.5
carry propagate adder	iPSC/860	8	8	20	641	34.734	572	723	0.2
carry lookahead adder	iPSC/2	1	1	20	1838	0.480	1838	1839	1.0
carry lookahead adder	iPSC/2	2	2	20	1204	22.379	1161	1264	1.5
carry lookahead adder	iPSC/2	4	4	20	725	21.727	654	772	2.5
carry lookahead adder	iPSC/2	8	8	20	905	22.675	848	960	2.0
carry lookahead adder	iPSC/860	1	1	20	253	0.430	253	254	1.0
carry lookahead adder	iPSC/860	2	2	20	223	12.729	175	240	1.1
carry lookahead adder	iPSC/860	4	4	20	310	15.547	277	348	0.8
carry lookahead adder	iPSC/860	8	8	20	648	38.163	587	788	0.4
carry lookahead adder	iPSC/2	1	1	64	14239	0.4068	14239	14240	1.0
carry lookahead adder	iPSC/2	2	2	64	8456	20.487	8438	8532	1.7
carry lookahead adder	iPSC/2	4	4	64	3009	33.630	2927	3095	4.7
carry lookahead adder	iPSC/2	8	8	64	2129	23.627	2089	2186	6.7

Table 10. Summary of Performance Data (cont.)

Circuit	Hypercube	LPs	Nodes	Input Vectors	Time (ms)	Std Dev	Min	Max	Speedup
carry lookahead adder	iPSC/860	1	1	64	2571	0.479	2570	2571	1.0
carry lookahead adder	iPSC/860	2	2	64	1542	11.091	1521	1565	1.7
carry lookahead adder	iPSC/860	4	4	64	685	68.793	628	944	3.8
carry lookahead adder	iPSC/860	8	8	64	770	67.189	697	1013	3.3
carry lookahead adder	iPSC/2	2	1	20	1476	27.575	1433	1542	1.3
carry lookahead adder	iPSC/2	4	1	20	1379	50.3008	1268	1494	1.3
carry lookahead adder	iPSC/2	8	1	20	2323	109.221	2094	2514	0.8
shifter	iPSC/860	1	1	20	452	0.379	451	452	1.0
shifter	iPSC/860	2	2	20	715	22.887	671	748	0.6
shifter	iPSC/860	4	4	20	1045	7.385	1033	1080	0.4
shifter	iPSC/860	8	8	20	1967	13.591	1921	1994	0.2
multiplier	iPSC/2	1	1	20	83261	0.740	83260	83263	1.0
multiplier	iPSC/2	2	2	20	32984	220.330	32382	33288	2.5
multiplier	iPSC/2	4	4	20	26220	667.850	25436	28040	3.2
multiplier	iPSC/2	8	8	20	39884	854.490	38296	41689	2.1
multiplier	iPSC/860	1	1	20	9270	0.712	9269	9271	1.0
multiplier	iPSC/860	2	2	20	7020	79.351	6729	7127	1.3
multiplier	iPSC/860	4	4	20	5388	113.202	5153	5575	1.7
multiplier	iPSC/860	8	8	20	8122	197.371	7840	8590	1.1
multiplier	iPSC/2	2	1	20	58247	503.57	57689	59741	1.4
multiplier	iPSC/2	4	1	20	71777	1007.47	69385	73717	1.2
multiplier	iPSC/2	8	1	20	177646	1491.65	175760	180523	0.5

Appendix F. *New Postprocessor Steps*

The postprocessor modifies the intermediate C code using the 10 steps Comeau described in his thesis (10), as well as two new steps. The first new step is to delete the following unnecessary function calls:

- `close_sigdict()`
- `m_int_type()`
- `m_real_type()`
- `m_real_type()`
- `m_signal()`
- `pop()`
- `push()`
- `read_input()`
- `rmtrrec()`
- `rptstats()`
- `rpterr()`
- `Start_Nonarray_Comp()`
- `sched()`
- `timer()`
- `tpop()`

The second new step is to modify every behavior instance's "function behavior" to report it's entity/architecture name if **MAPPING** is defined in VSIM and the boolean variable **mapping** is **true**. Each of these function declarations is of the form **Zxxxxxxx_xxxx(bi)**. Inside the function, after local declarations, the following is added:

```
#ifdef MAPPING
   if(mapping)
      printf("%s\n",  Zxxxxxxx_xxxx_trcbck);
#endif
```

Here is an example of a behavior instance function declaration prior to adding the new code:

```
static void
Z000002T_4440(bi)
BHP bi;
{
Z000002T_4112_struct *cd =
(Z000002T_4112_struct *)bi->data;
.
.
.
```

And here is the modified behavior instance function:

```
static void
Z000002T_4440(bi)
BHP bi;
{
Z000002T_4112_struct *cd =
(Z000002T_4112_struct *)bi->data;

#ifdef MAPPING
   if(mapping)
      printf("%s\n",  Z000002T_4440_trcbck);
#endif
.
.
.
```

Appendix G. *Key Source Code*

This Appendix describes some of the key source code necessary to implement parallel VHDL simulations with SPECTRUM. The code presented concerns interfacing VSIM and SPECTRUM. It is important to recognize that an event is logically equivalent to a signal change that is passed from one LP to another. A complete code listing is presented in a second volume.

G.1 vspec_init().

This routine builds a table of function pointers for SPECTRUM. Each function pointer represents the starting code for the simulation on each LP. For VSIM, *all* LPs start with the routine *startup()*. Therefore, every entry in *functions[]* is loaded with the address of *startup()*. Also, a call to *read_mapping()* is made so VSIM can determine which LPs are assigned which behaviors. Finally, a call is made to SPECTRUM's *lp_level_init()*, where SPECTRUM initializes and each LP calls *startup()*. Here is the code for *vspec_init()*, which is found in the file *vspec.c*:

```
void vspec_init()

{
   void (*functions[NUM_PROCS])();
   char *args[NUM_PROCS];
   char *argument;
   int i;

   /* initialize function pointers and lp #s as arguments */
   for (i = 0; i < NUM_PROCS; i++) {
      functions[i] = startup;
      argument = (char *)malloc(5*sizeof(char));
      sprintf(argument, "%d", i);
      args[i] = argument;
   }

   read_mapping();                    /* read in lpx.map file */

   lp_level_init(functions, args);
}
```

G.2 startup().

This routine, also found in *vspec.c*, is called by SPECTRUM after initialization. SPECTRUM passes each LP its LP number through *startup()*, and *startup()* calls *lp_init()* so SPECTRUM can initialize the VSIM filters. Finally, *vhdl_main()* is called in the intermediate C code so the circuit may be constructed. The source code for *startup()* is as follows:

```
void startup(lp_no)

   char *lp_no;

{
   sscanf(lp_no, "%d", &my_lp);          /* set global my_lp */
   free(lp_no);
   lp_init(my_lp);                       /* set up filter tables */

   vhdl_main();
}
```

G.3 send_signal().

This routine, also in the file *vspec.c*, is used to build an event out of a signal record, and call SPECTRUM's *lp_post_event()*, as follows:

```
void send_signal(this_signal_rec, dest)

   SIG_REC *this_signal_rec;
   int dest;

{
   struct event *new_event;

   new_event = (struct event *)malloc(sizeof(struct event));
   new_event -> from_lp = my_lp;
   new_event -> to_lp = dest;
   new_event -> time = *sim_time;
   new_event -> event = SIGNAL_CHANGE;
   new_event -> id = this_signal_rec -> sr_ptr -> id;
```

```
   new_event -> value = this_signal_rec -> value;
   new_event -> next = NULL;

   lp_post_event(new_event);
   free(new_event);

}
```

G.4 receive_signal().

This function passes received events from other LPs (via SPECTRUM) to VSIM. First, a call to SPECTRUM is made in *lp_get_event()*. This also activates the receive filter, shown later. If the filter passes *receive_signal()* a null pointer, then it returns to VSIM without posting a signal record into the local active list. Otherwise, the newly received event is converted into a signal record and posted directly into the active list. This function, shown below, is found in the file *vspec.c*.

```
void receive_signal()

{
   struct event *event;
   SIG_REC *new_sig_rec;
   SRP signal;
   int value;
   int time;

   event = lp_get_event();

   if (event != NULL) {
      signal = srrec_ptr[event -> id];
      value = event -> value;
      time = event -> time;
      MAKE_SIG_REC(signal, value, time);

      insert_sig_rec(new_sig_rec);
      free(event);
   }

}
```

G.5 null_post_fltr().

This is the filter used when VSIM sends an event to another LP. The filter is logically equivalent to AFIT's *chanclocks* post filter. For VSIM, the filter tracks the times a message was sent on each output arc. Also, when an event is sent to one LP, this filter sends a null message to all other output arcs. The post filter, found in the file *vfilt.c*, is follows:

```
void null_post_fltr()

{
   int i;

   /* update output channel time for this message */
   if (event_to_post -> event != NULL_MSG && NUM_OUT_LPS > 0) {
      for (i = 0; i < NUM_OUT_ARCS; i++) {
         if (OUT_ARCS(i) == event_to_post -> to_lp)
            output_ctime[i] = event_to_post -> time;
      }
   }

   /* send nulls to other lps */
   if (event_to_post->to_lp != my_lpid) {
      for (i = 0; i < NUM_OUT_ARCS; i++){
         if (OUT_ARCS(i) != event_to_post -> to_lp)
            send_null(OUT_ARCS(i), event_to_post -> time);
      }
   }
}
```

G.6 able_to_proceed().

This routine, found in the file *vfilt.c*, determines (1) at least one message has been received from every upstream LP, and (2) the next event in SPECTRUM's input queue is *less than* the safe time, as follows:

```
BOOL able_to_proceed()
```

```
{
   if (event_list == NULL)
      return FALSE;

   /* if haven't yet received a message from everybody */
   if (safetime() == -1)
      return FALSE;

   /* if still may get an earlier message */
   if (event_list -> time > safetime())
      return FALSE;

   return TRUE;
}
```

G.7 safetime().

This routine determines the minimum input channel time for all input arcs. It is found in *vfilt.c*, and lists as follows:

```
int safetime()

{
   int min_input_ctime = input_ctime[0];
   int i;

   for (i = 1; i < NUM_IN_ARCS; i++) {
      if (input_ctime[i] < min_input_ctime)
         min_input_ctime = input_ctime[i];
   }
      return (min_input_ctime);
}
```

G.8 send_nulls().

This function is used to send null messages to all downstream LPs prior to blocking for incoming messages (explained below in *null_get_fltr()*). The time stamp of the null messages is the

minimum of (1) the "low time" of the active list in VSIM, and (2) the safe time *plus* the output delay for the local LP. The code, found in *vfilt.c*, is as follows:

```
void send_nulls()

{
   int i;
   int safe_time = safetime();
   int vhdl_low_time = get_low_time();

   for (i = 0; i < NUM_OUT_ARCS; i++)
      if (OUT_ARCS(i) != my_lpid)
         if (vhdl_low_time < safe_time + LP_OUT_DELAYS(i))
            send_null(OUT_ARCS(i), vhdl_low_time);
         else
            send_null(OUT_ARCS(i), safe_time + LP_OUT_DELAYS(i));
}
```

G.9 null_get_fltr().

This is the filter used when receiving events from upstream LPs. First, if there are no upstream LPs, the function simply returns and VSIM continues. Otherwise, the filter gets a message (if *able_to_proceed()*), and returns it to VSIM if it is not a null message. The event is passed to VSIM by removing it from SPECTRUM's input queue and assigning it to SPECTRUM's global variable called *current_event*. If an event is not ready (not *able_to_proceed()*), then the filter determines if VSIM can continue without an event, i.e., if VSIM's *low time* is less than or equal to the safe time. The code, found in *vfilt.c*, is as follows:

```
void null_get_fltr()

{
   int vhdl_low_time = get_low_time();
   BOOL found = FALSE;
   EVENT *temp;

   if (NUM_IN_LPS == 0)
```

```
      return;

   while (!found) {
      if (able_to_proceed()) {
         if (event_list->event != NULL_MSG)
            found = TRUE;
         else {
            temp = event_list;
            event_list = event_list->next;
            node_trash_event(temp);
         }
      }
      else {
         if (vhdl_low_time <= safetime())
            return;
         else {
            send_nulls();
            while (!able_to_proceed())
            node_block_til_message();
         }
      }
   }

   current_event = event_list;
   event_list = event_list->next;
}
```

Bibliography

1. Ball, Duane and Susan Hoyt. "The Adaptive Time-Warp Concurrency Control Algorithm." *SCS Multiconference on Distributed Simulation*. 174–177. January 1990.
2. Banks, Jerry and John S. Carlson, II. *Discrete-Event System Simulation*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.
3. Bergman, Kenneth C. *Dynamic Spatial Partitioning of a Battlefield Parallel Discrete-Event Simulation*. MS thesis, AFIT/GCS/ENG, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.
4. Breeden, Thomas A., "AFIT Parallel VHDL Simulator User's Guide," 1992. User's Guide.
5. Chamberlain, Roger D. and Mark A. Franklin. "Hierarchical Discrete-Event Simulation on Hypercube Architectures," *IEEE Micro*, 10–20 (August 1990).
6. Chandy, K.M. and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions of Software Engineering*, *SE-5*(5):440–452 (September 1979).
7. Chandy, K.M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (April 1981).
8. Chandy, K.M. and J. Misra. "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, *1*(2):144–156 (May 1983).
9. Christensen, E., April 1992. AFIT Parallel Simulation Lecture.
10. Comeau, Ronald C. *Parallel Implementation of VHDL Simulations on the Intel iPSC/2 Hypercube*. MS thesis, AFIT/GCS/ENG/91D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
11. Daniel, David W. *Development of a Hardware Acceleration Engine*. MS thesis, AFIT/GCS/ENG, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.
12. DeGroat, Joe, et al. "AFIT VHDL Environment." *Proceedings-1988 Frontiers in Education Conference*. 324–329. 1988.
13. Dewey, Allen and Anthony Gadient. "VHDL Motivation," *IEEE Design and Test*, 12–16 (April 1986).
14. Fujimoto, Richard M. "Performance Measurements of Distributed Simulation Strategies." *Distributed Simulation 1988*. 14–20. 1988.
15. Fujimoto, Richard M. "Parallel Discrete Event Simulation." *Proceedings of the 1989 Winter Simulation Conference*. 1–34. 1989.
16. Hartrum, Thomas C., "AFIT Guide to SPECTRUM," 1992. User's Guide.
17. Hennessy, John L. and David A. Patterson. *Computer Architecture: a Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.
18. Hodges, Billy R., et al. "A Distributed Kernal for VHDL Simulation." *Proceedings of the IEEE 1990 National Aerospace and Electronics Conference*. 215–220. 1990.
19. Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

20. The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017. *IEEE Standard VHDL Language Reference Manual*, 1988.

21. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc, 1992.

22. Lipsett, Roger, et al. *VHDL: Hardware Description and Design*. Norwell MA: Kluwar Academic Publishers, 1990.

23. Mason, Tony and Doug Brown. *lex & yacc*. Sebastopol. CA 95472: O'Reilly & Associates, Inc., 1984.

24. McNear, Andrew E. *Improved Task Scheduling for Parallel Simulations*. MS thesis, AFIT/GCS/ENG/91D-14, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

25. Misra, Jayadev. "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, *18*(1):39–65 (March 1986).

26. Neelamkavil, Francis. *Computer Simulation and Modelling*. Dublin, Ireland: John Wiley & Sons, 1987.

27. Pritsker, Alan B. *Introduction to Simulation and SLAM II*. West Lafayette, Indiana: Systems Publishing Corporation, 1986.

28. Proicou, Michael Chris. *A Distributed Kernel for Simulation of the VHSIC Hardware Description Language*. MS thesis, AFIT/GCS/ENG/89D-14, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

29. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers*. Mc Graw-Hill, Inc., 1984.

30. Reynolds, Jr., Paul F. "A Spectrum of Options for Parallel Simulation." *Proceedings of the ACM Winter Simulation Conference*. 1988.

31. Reynolds, Jr., Paul F. "Comparative Analyses of Parallel Simulation Protocols." *Proceedings of the 1989 Winter Simulation Conference*. 671–679. 1989.

32. Reynolds, Jr., Paul F. and P.M. Dickens. "SPECTRUM: A Parallel Simulation Testbed." *Proceedings of the 4th Annual Hypercube Conference*. 1989.

33. Van Horn, Prescott J. *Development of a Protocol User's Guideline for Conservative Parallel Simulations*. MS thesis, AFIT/GCS/ENG/92D-19, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

34. Zhang, Guoqing. "Partitioning and Transformation of VHDL Models for Distributed Simulation." *6th Workshop on Parallel and Distributed Simulation (PADS92)*. 203–205. 1992.

Vita

Captain Thomas Andrew Breeden was born on October 14, 1961, in Punta Gorda, Florida. He graduated from J.M. Tate High School in Pensacola, Florida, in 1979 and enlisted in the Air Force in 1981. In 1988, he received a B.S. Electrical Engineering degree, with high honors, from the University of Florida. He was commissioned a Second Lieutenant on September 29, 1988. Captain Breeden was assigned to the 6555th Aerospace Test Group, Cape Canaveral Air Force Station, as a launch network controller before his entry into AFIT in May 1991.

Permanent address: 8567 Chemstrand Rd.
Pensacola FL 32514

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
PARALLEL SIMULATION OF STRUCTURAL VHDL CIRCUITS ON INTEL HYPERCUBES

5. FUNDING NUMBERS

6. AUTHOR(S)
Thomas A. Breeden, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/92D-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA
3701 North Fairfax Drive
Arlington, VA 22203
(703) 696-2298

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many VLSI circuit designs are too large to be simulated with VHDL in a reasonable amount of time. One approach to reducing the simulation time is to distribute the simulation over several processors. This research creates an environment for designing and simulating structural VHDL circuits on the Intel iPSC/2 and iPSC/860 Hypercubes. Logic gates and system behaviors are partitioned among the processors, and signal changes are shared via event messages. Circuit simulations are run over the SPECTRUM parallel simulation testbed, and the null-message paradigm is used to avoid deadlock. Structural circuits ranging from forty to over one thousand logic gates are correctly simulated. Although no attempt is made to find *optimal* partitioning strategies, speedups are obtained for some configurations.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Simulation, VHDL, Circuit Simulation, Intel Hypercube	167
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL